April 2009

a PC based SDR platform with dynamic reconfiguration

Alexander Valdemar Camilo
Worcester Polytechnic Institute

Francesco Paul Bivona
Worcester Polytechnic Institute

Reconfigurable Software Defined Radio Platform

A Major Qualifying Project Report:

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the Degree of Bachelor of Science by

_____

Francesco Bivona

_____

Alexander Camilo

Date: March 25, 2009

Approved:

_____

Professor Xinming Huang, Major Advisor

_____

Professor Alexander Wyglinski, Co-Advisor

Abstract

The goal of this Major Qualifying Project is to provide the framework for integration of a Virtex series field programmable gate array (FPGA) into GNU Radio, allowing GNU Radio to have control over both FPGA and non-FPGA components of the pipeline. In this report, we address the following: our research into the which FPGA series would be most beneficial to our project, an outline of the evolution of our design over the course of the past 21 weeks, and a summary of the final outcomes in various subsets of project development.

Executive Summary

The concept of modular reconfigurable radio has spanned decades, beginning humbly with hardware-based reconfigurable radio such as the Joint Tactical Radio System and the Speakeasy Project, and progressing to software-defined radio application programming interfaces (APIs) such as GNU Radio [1][2]. Our project seeks to take this a step further, and integrate reconfigurable hardware (a field programmable gate array) into a software-defined radio pipeline.

To this end, we conducted significant research and development with regard to FPGA architecture for SDR applications. This consisted of attempts at implementing a bidirectional peripheral component interconnect express (PCIe) interface and basic dynamic reconfiguration using the internal configuration and access port (ICAP) interface. Eventually, the decision was made to streamline the architecture using the XILINX Embedded Design Kit (EDK)
[22]. At this stage, we have a functional bidirectional PCIe interface (with the driver still being a work-in-progress), and dynamic reconfiguration via ICAP functioning in a test bench environment.

Additionally, we have conducted experimentation into the application of GNU Radio to specific tasks. This began with simple frequency modulation (FM) radio transmission, and then moved on to more complex digital waveforms and information transfer between two computers. This experimentation also accommodated for the presence of an FPGA (in tandem with simultaneous FPGA architecture development), either in the form of passing a data through unhindered or

being used as a signal generator. Additionally, an investigation into the inner workings of GNU Radio signal processing blocks was carried out.

Finally, our focus was directed toward FPGA integration with GNU Radio. This involves finalizing filter architecture for the FPGA using DSP48e slices, final debugging of the FPGA's hardware device driver, and writing up a GNU Radio pipeline to wrap FPGA and PC signal processing components together to complete a task.

The end product of this project consisted of a functional GNU Radio pipeline, implementing the FPGA to carry out filtering. We were able to synthesize a lowpass filter on the FPGA, followed by a raised cosine filter when the hardware driver was debugged with the lowpass filter example. File I/O still remains slightly buggy, and the destination file for the output did not store information. However, GNU Radio was capable of controlling and making use of the FPGA in all other respects, and functionality was confirmed by displaying filter output in the terminal. While integrating an FPGA (independent of the front-end's FPGA) has been accomplished before, this is the first seen of it within a widely-used environment such as GNU Radio. The current platform sets the framework for future research into implementing dynamic reconfiguration of the radio pipeline.

Table of Contents

1. Introduction

Software-defined radio (SDR) is a very active field in the world of digital signal processing. It is used in a variety of scenarios, where either mobile or easily adaptable systems are required. Thus, we have chosen this field, since it is a useful subject to be knowledgeable about and to contribute data toward. For the purposes of this project, we intended to integrate a field programmable gate array into an otherwise strictly software-composed pipeline. This would delegate hardware resources away from the host computer, without losing the reconfigurable nature of software. To prepare for this, we implemented FPGA loopback in a variety of GNU Radio applications, and ultimately implemented filtering of signals on the FPGA. In future work, it would be possible to adapt the FPGA component of the pipeline such that it can be dynamically reconfigured, which is less complicated to implement on reconfigurable hardware than in software.The primary motivation for this project was to accelerate the development of software defined radio, integrating hardware that is just as easily reconfigurable as the software components in the pipeline. Currently, FPGAs have been integrated into software defined radio platforms, but it has been limited in terms of which radio frequency (RF) front-end is used, and which API controls the hardware. While tested only with DD [21] a program intended for copying and pasting data, which we used for debugging) and GNU Radio, our design can be controlled by any existing SDR API that supports UNIX file I/O, and can interact with any RF front-end the current API supports, since it runs independent of the front end and the API draws the connections between the two. We have established a framework with which future researchers could implement further functions, such as dynamic reconfiguration.

2. Literature Survey

There has been prior research and development put behind the concept of reconfigurable radio. This extends back as far as the mid-‘90s with military investment in modular radio components [1] [2]. It was revolutionized with the concept of software-defined radio, which minimized proprietary hardware and implemented most signal processing within a computer. This was brought into the mainstream with the Spectrumware project, later to become GNU Radio [5] [6]. Since then, some have investigated the possibility of a combination of the two methods, implementing an FPGA in collaboration with a PC for digital signal processing. It maintains the reconfigurable nature of software while still having more resources at its disposal than a non-dedicated personal computer. There are even dynamic reconfiguration possibilities which surpass those of software-only pipelines.

2.1 Origins of Reconfigurable Radio

Prior to the advent of today’s FPGAs and software-defined radio, the concept of radio that one can configure to be compatible with multiple mediums was still considered. However, it was more intricate. With the lack of ability to alter the data pipeline with some lines of code, data processing modules for different tasks were just that – physical “modules” that had to be stored, installed, and swapped out as the application saw fit. Such was the concept behind the Joint Tactical Radio System (JTRS) and the Speakeasy project [1][2]. They are multiprocessor systems (JTRS using Pentium cores, and Speakeasy using four TMS320C40 processors) that require extensive custom hardware to perform many of the signal processing tasks presented by the varying military communication conventions available [1] [2].

2.2 The Introduction of Software Defined Radio

Software defined radio is the replacement of analog components in a radio pipeline, at the very least starting at the IF stage and continuing from that point onward [3]. However, ideally all hardware should be replaced with software from the antenna onward [3]. RF still cannot yet practically be processed with general-purpose processors. Furthermore, software radio is overall slower than implementing it in hardware for certain applications (such as modulation), so implementing configurable hardware such as an FPGA at certain points would improve performance without limiting the configurability and multimode operation that are the most significant benefit of software-defined radio [3]. Application-specific integrated circuit (ASIC) designs would be able to accomplish similar tasks, but reconfigurability is at a minimum since ASICs are not readily reconfigurable after manufacture. There is one area where ASIC outshines an FPGA – power consumption. As a general rule, state-of-the-art ASIC designs are more power-efficient than equivalents implemented on an FPGA. An illustration of this is the measurement of power consumption during the implementation of numerous counters, in one case on an Altera FPGA, and in another with an ASIC. See Table 1 for the results.

Number of active counter blocks	1	16
ASIC power (mW)	0.9	9.9
FPGA power (mW)	53.6	112.4

Table 1: Power consumption on different numbers of both ASIC and FPGA counter blocks. The FPGA exceeds the ASIC in power consumption by orders of magnitude [4].

This primarily limits FPGA use to stationary tasks rather than mobile ones. ASIC is still superior in mobile applications. However, the benefits of applying an FPGA when increased power consumption is tolerable are not to be ignored: Ease of design and manufacture, fewer discrete components – these are attained with the implementation of FPGAs [5]. With significant

streamlining and development, cost can be reduced and standardized APIs for SDR components can be developed.

2.3 Spectrumware and GNU Radio [6] [7]

The Spectrumware project is the predecessor of modern software defined radio. Spectrumware later proceeded to branch off into two directions: Vanu Radio – the commercialized version of the Spectrumware architecture – and GNU Radio – the open-source standard for any research application involving the Universal Software Radio Peripheral (USRP). Spectrumware and its descendants are actually far more economical than hardware module-based designs. Using GNU Radio as an example, the software is open-source and can be used with most modern personal computers (if they are running Linux). The only additional hardware required is a front-end, usually satisfied by the USRP. This hardware costs $700 [8]. This is an insignificant amount when compared to hardware that required a budget from the United States military to produce. With the advent of more powerful microprocessors, more and more of the system could be simulated by software. Thus, the pipeline is reduced to minimum transmitter/receiver hardware (a small FPGA is used for initial signal processing in the USRP in the case of GNU Radio, and similar proprietary hardware with Vanu radio) and an extensive API for simulating signal processing modules that once required dedicated hardware. The system still has its drawbacks. It is very CPU-intensive, while an FPGA-assisted platform would take much of the burden off of the microprocessor. Additionally, under normal circumstances, a GNU Radio pipeline must be stopped for some time, however short, in order to be modified. An FPGA implementing dynamic reconfiguration would provide the ability to modify the pipeline mid-operation. This will prove useful in tandem with existing GNU Radio applications.

2.4 FPGAs and Dynamic Reconfiguration

FPGAs have been implemented in a software-radio environment to some extent for a significant amount of time – the FPGA integrated into the USRP is a good example of this. The USRP includes an Altera Cyclone, which is significantly smaller and less powerful than the Virtex-5 we intend to implement [9]. Furthermore, it is incapable of dynamic reconfiguration. However, this is acceptable since it is only meant to handle high sample-rate processing involved in transmission and reception [9]. Additional hardware is included in the form of daughterboards, which contain dedicated transmitter/receiver/transceiver hardware [9]. They are small and easily transferable depending on the desired application, ranging from DC-30Hz transmitters/ receivers to 2.4 GHz/5 GHz dual band transceivers [9].

There are even instances where dynamic reconfiguration of the pipeline with an FPGA is proposed. FPGAs have generally been applied to the SDR pipeline in different ways, or applied to a different task entirely. For instance, the 7142 Virtex-4 software radio PMC/XMC Mezzanine takes the place of both the radio frequency (RF) front-end and the modulation segment of the pipeline [10]. This makes it more limited, since its RF front-end cannot be as easily replaced if a certain task calls for it. Our design uses the USRP as an RF front-end, but this can be later swapped with minor modification, without rendering a portion of the FPGA peripheral outdated.

2.5 FPGAs and Software Defined Radio

In 2005, a plan was proposed (but not implemented) to use an FPGA in a similar manner to our platform, using partial reconfiguration to modify the pipeline as needed. Research into the subject was extensive, assigning different levels of reconfiguration desirable within an active

pipeline: standard (GSM, 802.11g, etc.), mode (DSSS, FHSS, etc.), and service (bandwidth, etc.) switching. The pipeline's actions were also broken down into categories based on hardware requirements - Modulation (FPGA, Digital Signal Processor), Data Handling (microprocessor, RAM), and Coding (DSP) [11]. These tasks should be divided into three hierarchical layers – the overall communication class, functions within this class (which could be modified for specific applications of a certain communication class), and the most internal layer, including the contents of a particular function [12]. Depending on the changes that need to be made from one pipeline to another, modifying a certain hierarchical layer would make the reconfiguration more efficient than modifying the pipeline on the most intricate level [12]. Modular design simplifies the implementation and modification of a pipeline. A collection of pre-designed modules capable of being swapped out is easier than having to rewrite a segment of one large piece of code, both in terms of the user's understanding and the complexity of reconfiguration. Reconfiguring on this scale is a known as reconfiguring on the functional level, and is in general what we have chosen to implement [13].

There are a few concerns that must be taken into account when implementing a design on a dynamically reconfigurable FPGA. First, one must take into account the reconfiguration time. Reconfiguring the FPGA directly, it would be split into slices of the same pipeline, and any modification to slices would interrupt the flow of data until the new slices were written [14]. This can be alleviated by making a copy of the entire pipeline and modifying that [14]. Copying the pipeline allows the reconfiguration to take whatever time is necessary to overwrite specified portions, and the data stream is only interrupted while switching from the original pipeline to the modified pipeline [14]. This form of reconfiguration is something referred to as "merge

configuration" [15]. This also allows for slices that do not span the entire length of the FPGA, given a properly configured communication link between them [15]. However, it requires more resources than direct reconfiguration, since at any reconfiguration time there must be two instances of the pipeline on the FPGA [15].

The communication link, or the means by which the blocks will communicate with each other, is another issue which must be addressed. On the surface, the solution appears simple: we must implement a bus macro – a set of inputs and outputs that are held constant between all slices regardless of configuration, and an arbiter directing IO to/from each slice. However, we must be cautious, because the implementation of a bus macro is platform dependent. Work must be done somewhat from scratch, which is a significant effort. One idea for an adaptive bus macro design is as follows: HIBI (Heterogeneous IP Block interconnection) links together blocks with a variety of different interfaces (open core protocol (OCP), first-in-first-out (FIFO), direct memory access (DMA)) [16]. This is a very adaptive design, even though it is not developed for the Virtex-5 [17]. It would be useful in that there are likely many modulation task blocks already created, and this allows for a variety of IO interfaces to be interconnected [16]. A platform-specific bus macro would not be nearly as versatile, and rely on blocks created specifically to function with that macro [16]. This is worth looking into, should we find our initial bus macro inadequate [16]. What makes our design significantly different is that the "arbiter" will likely be the host PC's microprocessor itself (along with any other data management, etc.), rather than an FPGA block. In all of Delahaye's work, there is no mention of using device-nodes as a means of hardware interface with preexisting SDR APIs such as GNU Radio. This work comes closest to

matching our own (at times), and evidence to the contrary has not yet been unveiled, device-node IO implemented in this manner is new to the concept of linear SDR pipelines.

The final significant decision that needs to be made is whether to use internal or external configuration [17]. "External" configuration involves a CPLD or microprocessor directly overwriting the FPGA's contents, with no internal intervention [17]. However, the Virtex-5's ICAP interface allows for what is called "internal" configuration, or auto-configuration [17]. Issuing commands and data to the ICAP allows for the FPGA to largely rewrite itself, with less equipment involved (only the host PC to issue commands/information) [17]. For our application, it is far more effective to implement internal reconfiguration, since the resources are already there. The ICAP does limit us to partial reconfiguration, but that is acceptable. We're not changing the structure of the entire FPGA (size/number of slices) with reconfiguration, only the contents of the slices. FPGA structure remains constant unless a different chip is used. Furthermore, partial reconfiguration requires less overhead than a complete re-flash of the FPGA [17].

2.6 Platform Applications

There are numerous applications for SDR in a mobile environment – cellular phones, mobile wireless, and so on. The distinction should be made here that our design is *not* meant for such tasks. The design method taken is simply not well-suited. It is critical to keep power consumption to a bare minimum in a mobile environment, and FPGAs are renowned for their power consumption, the Virtex-5 being no exception. This is in addition to the USRP requiring its own power supply, and, of course, the host PC's power. A. Dejonghe defines the acceptable rate of power consumption for a digital baseband platform to be 100 million operations per

second (MOPS)/mW [18]. This is nowhere near the current level of power efficiency for modern FPGA technology at all, let alone an FPGA implementation. Referring back to Table 1, even implementing 16 counters, each of which only use a few components, exceeds desired power consumption. Given the components, effective mobile use of SDR as a reconfigurable platform would be infeasible at this time. It is worth mentioning, however, that "mobile" in this case does not include being inside vehicles, or connected to some other power source which gives our platform access to surplus power while still technically being capable of changing location. In such an environment is where our platform would thrive, particularly in military applications. Of course, it would also work in a stationary, desktop environment, for research or other purposes.

Chapter Summary

Some of the branches of software defined radio mentioned here, namely FPGA integration and dynamic reconfiguration, are within the realm of this project or in the near future work of the project. However, some such as cognitive radio are a bit far off in terms of development with respect to our project. Still, having knowledge of the possibilities does lend itself to accelerating progress on the path toward said possibilities, and was thus worth investigating. This project does not comprise the entirety of software-defined radio development. Due to limited time and resources, this project is mainly focused on integrating an FPGA into an SDR environment.

3. Initial Design Concept

Our design is an attempt at integrating an FPGA peripheral of our choosing into a software-defined radio pipeline. Time permitting, we also intended to implement some means of dynamic reconfiguration, and have plotted out strategies for such an endeavor. These remain unimplemented, and are a recommended continuation for anyone who sees fit to pursue similar goals. At the very beginning of our project, some key decisions had to be made. Most significant among these were the choice of the FPGA we were to use, and the choice of interface between PC and FPGA peripheral.

3.1 Choice of FPGA

The Virtex 5 series of FPGAs can be broken into 4 different classes of device each targeting a specific type of design challenge. The 4 families are the LX, LXT, SXT, FXT. The T means that the FPGA has hardware for high speed transceivers which are required to implement protocols such as SATA and PCIe. The family is additionally broken up into the LX, SX, and FX, groups. The LX and LXT FPGAs contain a large amount of generic CLBs and little dedicated hardware. The FXT family contains embedded hard-cores for SOC type designs. The DSP variant of the family is the SXT with 6 colums of 49 DSP48e slices. We believe that these DSP blocks which consist of a 25x18 bit multiplier and 48 bit accumulate register would be extremely beneficial for our application, because they wouls allow us to make high-speed implementations of common signal processing blocks.

3.2 Possible Choice of board

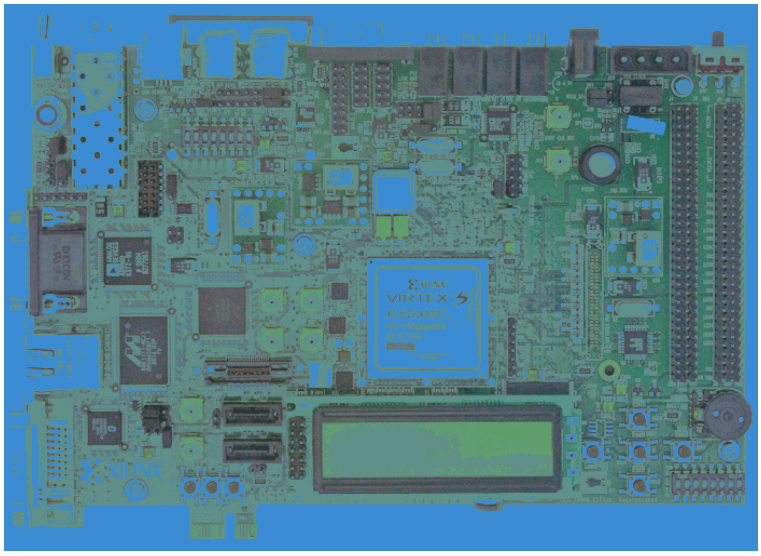

Figure 1: ML506 FPGA development board featuring a VIRTEX 5 SXT FPGA and a 1x PCIe connector.

The ML506 development board from Xilinx would be well suited to our project for several reasons. The FPGA on this board is a XC5VSX50TFFG1136 which is a Virtex 5 SXT series FPGA. This FPGA provides us with 288 DSP blocks which would simplify the design of pipeline modules by replacing commonly synthesized hardware with real hardware blocks. In addition to the DSP slices the FPGA supports multi gigabit transceivers giving it the capability to use PCIe. The development board includes interfaces for all of the feasible methods of interfacing the device with the host computer. These interfaces include PCIe, USB, and gigE. As well as having the proper interfaces and resources, this board's price point is low when compared to other development boards we have found.

3.3 Bus Interfaces

Ultimately, the exact interface does not matter if we structure the code correctly and view the interface as a pipe on a higher level.

3.3.1 Option 1: USB

Universal Serial Bus (USB) is one option we could have pursued for interfacing our FPGA with a PC and RF frontend. While more ideal for external hardware than a peripheral card that is meant to be integrated into the PC, it is sufficient. USB bandwidth peaks at 60 Mbps, of which only 10% depending on transmission method would be used. The ideal way to go about the interface with USB is to make use of bulk transfers. The performance is heavily dependent on other traffic present on the bus.

If there is no other traffic, the bulk transfers can make use of up to 90% of the bandwidth. With traffic, however, bulk transfers receive the lowest priority and thus suffer from severe latency problems. There are alternative methods of data transfer with USB (i.e. isochronous transfer) which have higher priority, but always have a lower maximum bandwidth percentage. What a USB interface would require is an entire bus dedicated to the SDR platform, using bulk transfers to handle data. Additionally, the USRP only supports a gigE interface, not USB. So using a USB interface for our peripheral card for the sake of uniformity becomes a moot point if the user decides to use the USRP2 rather than the USRP.

3.3.2 Option 2: PCIe

The PCIe interface option provides several advantages. Latency becomes less pronounced, with PCIe generally closer to the system memory and CPU on the PC. With a PCIe interface, we can also take advantage of the conventions of PCIe, which allow for DMA and bus mastering to communicate between peripherals independent of the PC. Another significant benefit is the greater bandwidth which, for PCIe, is 2.5 Gbps or higher minus overhead. On the other hand, USB provides 60 Mbps minus overhead. While the USB bandwidth is acceptable, PCIe proves to be more promising. Additionally, PCIe leaves us with a platform that is more expandable. The device can be compatible with a wider array of RF front-ends than the USRP due to increased theoretical bandwith. Front-ends are already easily interchangeable, since the ultimate purpose of our peripheral card is to be reconfigurable, and information accepted and given by front-ends is already similar from front-end to front-end. With the USRP's USB interface, the latency and bandwidth advantages of PCIe are only seen between peripheral card and PC. However, if an RF front-end that also uses PCIe (or a faster interface such as gigE) was introduced in place of it, the bus mastering and DMA allow for more efficient transfer of data, so front-end and peripheral would be able to communicate with each other with out the overhead of the CPU.

Admittedly, PCIe adds a certain complexity to the interface compared to USB. While USB uses periodic frames with packets for control, the PCIe interface operates with streams of packets with a layered protocol stack. However, the length of time developing the software side of the interface is greatly reduced by the presence of pre-existing code and IP modules. XILINX software tools contain the code required, and even have a wizard for setting up the Virtex-5

FPGA as PCIe block RAM, in addition to having a PCIe-processor local bus (PLB) bridge, which simplifies DMA access. This overcomes any difficulties the added complexity present.

3.4 Dynamic Reconfiguration

3.4.1 Hardware

XILINX's Virtex series and the Spartan-6 support partial reconfiguration, where a portion of the FPGA can be reprogrammed while the other portion of the FPGA is still operational. The Virtex-2, 4, and 5 series FPGAs support dynamic reconfiguration through several interfaces, Joint test action group (JTAG), Serial Perepheral Interface (SPI), and SelectMap (accessable through the internal configuration and access port) interfaces. SPI and SelectMap are external interfaces that can be used, which means that a bitmap can be applied from iMPACT (Xilinx's programming utility) or by an external controller.

The ICAP configuration interface allows for self-reconfiguration. Synthesized hardware may interface to the ICAP interface, which appears as a bidirectional synchronous parallel bus of varying with depending on the family, and issue the sequence of commands present in a bit stream to change routing, configurable logic block (CLB) configuration, and block RAM (BRAM) contents. The more recent series of FPGAs such as the Virtex-4 and Virtex-5 series support wider bus widths allowing for faster reconfiguration.

3.4.2 Software

The way dynamic reconfiguration is accomplished is through partial bitmaps. Instead of containing configuration information for a whole design these partial bitmaps contain configuration information for only a few CLBs allowing for a large portion of the FPGA to remain unmodified while reconfiguration takes place.

Communication between the hardware included in these partial bitmaps is accomplished through a bus macro. The bus macro defines an interface between the existing static logic and the newly configured logic, much such as a connector or socket. XILINX provides tools to facilitate dynamic reconfiguration by allowing low level changes to be made to bitmaps. This tool is called Plan-Ahead.

3.5 Proposed Methods of Dynamic Reconfiguration

After doing some research as to how dynamic reconfiguration functions on XILINX based FPGAs, we have come up with several possible designs that would achieve dynamic reconfiguration. These designs vary in complexity and cost and are organized starting with the easiest to implement and ending with the hardest yet most interesting design. While dynamic reconfiguration was not implemented, we still devoted time to the background study, which may prove useful to future researchers. Thus, these sections remain for future reference.

3.5.1 Single Module Dymanic Reconfiguration

In this design we use a single module with a parallel bus macro for incoming and outgoing samples connected to some mechanisim designed facilitate the transfer of data to and from the host over our chosen interface. In addition to this mechanisim the bitmap would contain enough hardware to grab a configuration file from the host system via an interface such as USB, gigabit Ethernet (gigE), or PCIe and change the configuration of this one "module" shown in figure 2 via its internal ICAP interface.

Using this method of configuring the FPGA we could have a bitmap for various different pipelines and we would have the capability to change our current pipeline. The downside of this approach is the pipelines themselves would be fairly static. We would not be able to alter the structure of individual blocks dynamically.

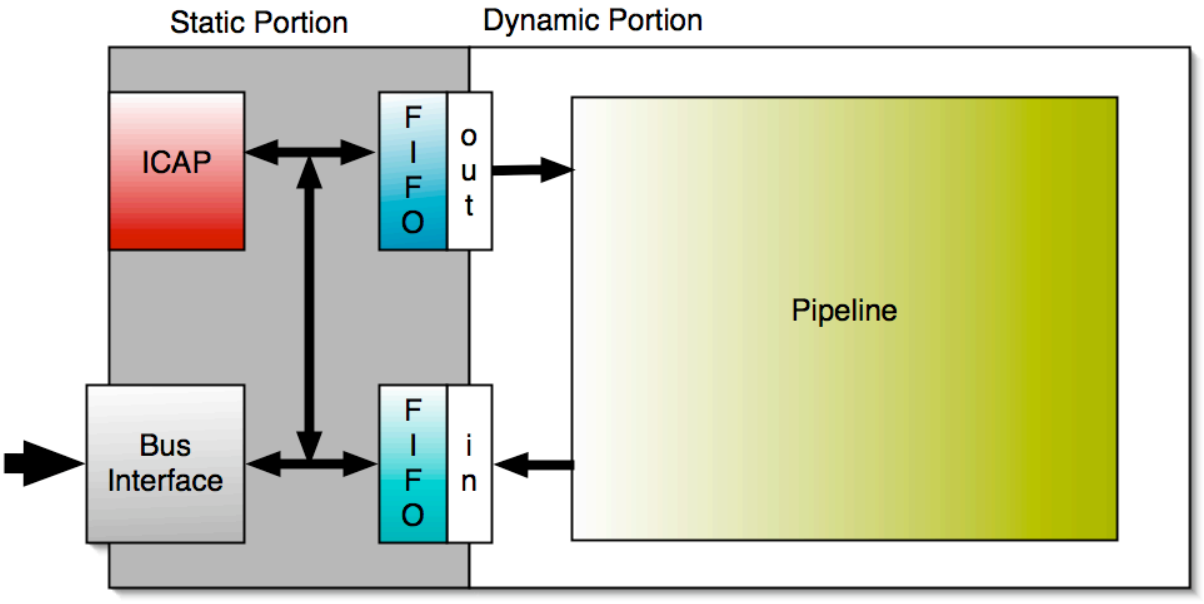

Figure 2: Dymanic reconfiguration where a module is an entire pipline.

Since this design does not vary very far from XILINX's documentation and the beaten path and it should be fairly easy to implement. The complex part would be generating a few sample bitmaps and writing a loader that would apply these bitmaps. The benefit of this is that the computer can change the pipeline directly from within their signal processing environment.

3.5.2 Multiple Difference Bitmaps

Another approach that would give us more control over the actual structure of the SDR pipeline while still remaining low in complexity would be to set aside a certain amount of slices for blocks and generate difference bitmaps for a given module in these different slots as seen in figure 3.

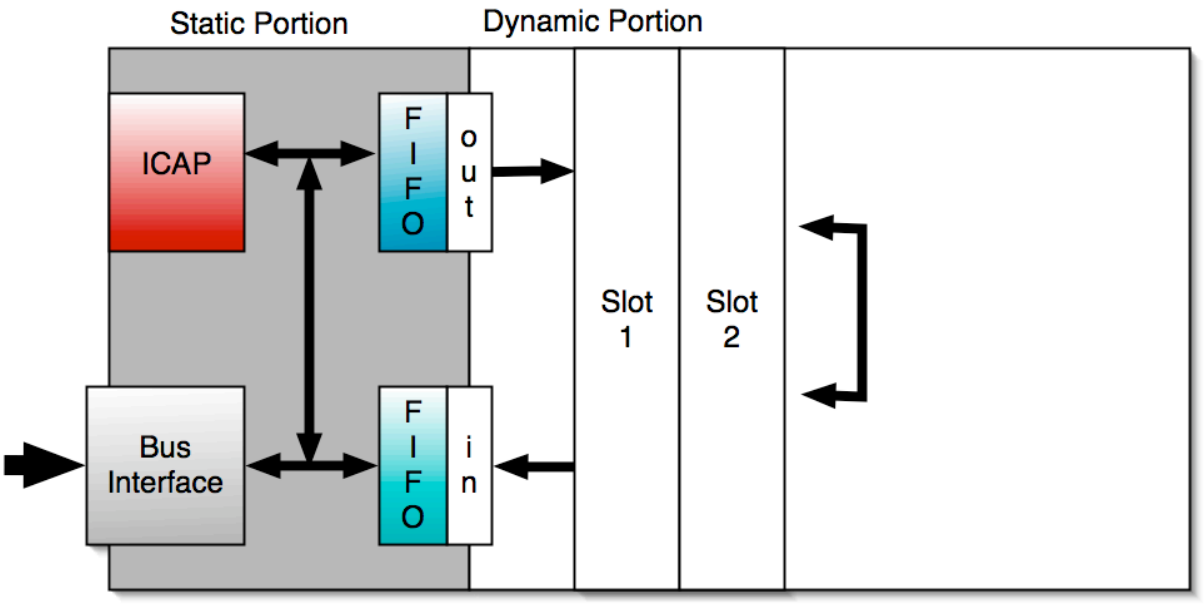

Figure 3: A diagram outlining the process of dynamic reconfiguration using bitmaps.

Communication between modules would be handled by bus macros on the boundaries between slices. The benefits of this approach are increased control over the structure of the SDR pipeline. We could switch between BPSK and QPSK depending on the quality of the channel for example. The downside of this approach would be that if our pipeline had 10 "slots" we would need 10

bitstreams for each module where each bitstream would be that module in that portion of the FPGA. A QPSK module would have a “in slot 1” bitmap and an “in slot 2” bitmap. This is due to limitations in XILINX’s synthesis and PAR (place and route) tools and in the hardware of the virtex 5 itself. An Analogy in terms of software would be machine code with absolute jumps (as opposed to relative jumps). Such code could only be located at an initiall address in system specified at compile time and different compiled versions would be needed if different starting vectors where desired.

3.5.3 Independent Application

The final approach would be to gain a better understanding of the structure of the configuration bitmap such that we could have an application keep and modify this bitmap representation of the FPGA’s structure and generate a dynamic bitmap that would be used to reconfigure the device on a very fine level. For example, a program could take a generic template of a module and make a modification on the routing and configuration of the internal structure of the FPGA to incorporate it into the SDR pipeline. This approach could potentially be dangerous and lead to damaged hardware because it would involve reverse-engeneering XILINX’s proprietary bitmap format.

Chapter Summary

In this section, we have outlined the various possible ways that we could have approached the hardware design and software interface problems. A desired bus interface to develop around, method to implement dynamic reconfiguration, and a way of integrating our FPGA peripheral into GNU Radio are all required at first. Through the evaluation of hardware, we have settled on

PCIe for the bus interface, single module dynamic reconfiguration, and the device node interface to tie the FPGA and GNU Radio together.

4. Project Development: GNU Radio

GNU Radio is a multifaceted, complex tool which we have used throughout the majority of this project. We performed initial test runs of tasks such as FM transmission with GNU Radio on its own, and later integrated our FPGA into the pipeline. We also investigated the lower-level functionality of GNU Radio, and made an attempt at creating a signal processing block. Toward the end of the three-term project, we attempted more complex tasks such as differential binary phase-shift keying (DBPSK) modulated digital transfer of data via TCP/IP tunnel and direct transfer. We integrated the FPGA into the latter with a loopback configuration to show our progress on hardware integration within GNU Radio up to that point.

4.1 Integrating the FPGA and a Host Application (GNU Radio)

GNU Radio is a Python- and C-based API that is used for a wide variety of software-defined radio applications, such as amplitude modulation (AM) transmission, frequency modulation (FM) transmission, packet handling, filtering, et cetera. In particular, portions of the API are dedicated to the operation of the USRP. Since we are using the USRP as our analog front-end, this provides an important glimpse into how software-defined radio tasks are carried out in terms of the host PC. On a general level, the API functions by having component classes (filters, etc.) and functions coded in C, with Python code segments connecting the blocks. Most Python segments literally contain only configuration for the blocks required, then several instances of calling a 'connect' function on itself to construct the order of the pipeline. The FPGA peripheral will ultimately do the intensive work for our platform; much of the pipeline will not be implemented in GNU Radio. However, the basic structure that it provides for its pipelines can give us a framework for our own.

An example is the file `am_rcv.py` included with the GNU Radio API. This file, as its title suggests, sends and receives amplitude modulation (AM) frequencies. The first block is a digital down converter (DDC), used to select a narrower band of frequencies to receive from the wide array of AM frequencies available. Then the magnitude of the DDC output is taken (MAG) to have a strictly real input, and the volume is scaled down (VOL). This is followed by deemphasis (IIR) and an audio filter (FIR) to reduce noise, and the data finally gets output to the audio sink (in other words, it is played on the computer's speakers). The same process could, theoretically, be followed in reverse to transmit on AM frequencies. A visual representation of the AM pipeline can be found in figure 4.

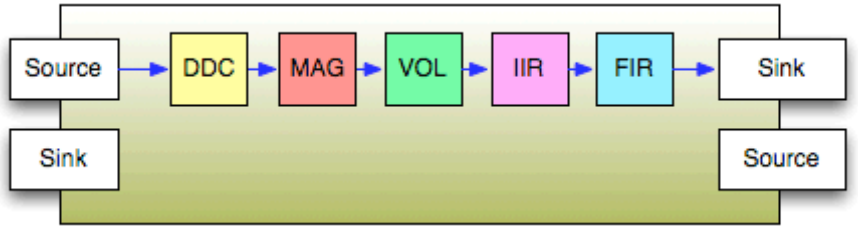

Figure 4: An AM radio pipeline from GNU Radio. This highlights the top-level structure of a typical GNU Radio pipeline.

In addition to introducing template ideas, GNU Radio also provides infrastructure for the management of data source and sink in our design. GNU Radio already contains functions for creating a file as a data sink and for using a file as a source. With this format, the FPGA can send/receive data with general UNIX file IO. Even though it appears to be the case on the software level, the data is not actually stored in a file. The "files" in question are actually calls to

functions that work in kernel space, directing anything written to them to addresses representing the desired peripheral IO and pulling anything read from them from the same addresses. File IO is notorious for being slow under normal circumstances, but this is due to the medium used. If located on the PC's hard disk (as per usual), file IO is slow. However, the file IO API itself is not.

GNU Radio's top-level applications are implemented in Python, while our FPGA is configured using a bitmap synthesized from Very High Speed Integrated Circuits Hardware Development Language (VHDL) or Verilog. Both are expected to perform within the same pipeline. This distinction in languages begs the question: how do the two languages interact with one another? Within GNU Radio exists wrappers which convert the C objects in the library into Python-friendly code. While we have no such wrapper to integrate the FPGA into code manageable by Python, there is an elegant option. The concept of device-nodes exists in UNIX, and transcends both programming languages. The FPGA will be configured using an API that is set up apart from GNU Radio. This will consist of blocks created within the XILINX ISE that are combined by the API and applied to the card. Our drivers will establish device-nodes for the FPGA card's IO and configuration, requiring a request to the kernel for information regarding where the FPGA expects to receive configuration data, receive input, and deliver output. Input, output, and configuration streams will be functions that are expressed in UNIX as files that actually store information in a buffer before moving it to its specified location. With this established, the FPGA (and therefore all of the pipeline elements within) will be treated as a file by the rest of the computer.

The end result (once drivers have been written) is that configuration of the pipeline occurs in two phases. First is the configuration of the FPGA. As long as components loaded to the FPGA are contained within the API, no use of XILINX ISE would be necessary. It would only be a matter of appending preexisting files to the configuration "file", with specification of which portion of the FPGA it should be written to. The driver will then load the file to the expected location on the FPGA. This will be where all blocks expected to be implemented by the FPGA will be configured, primarily modulation tasks (encoding, filtering, mapping, etc. - this will vary from standard to standard). This will be followed by configuration of the data management portion of the pipeline (multiplexing, etc.), which is best dealt with by the microprocessor [21]. Any tasks the microprocessor is expected to accomplish will be managed by manipulating GNU Radio conventionally.

The USRP is connected via the PCs USB and not through the FPGA, so the microprocessor can still access the USRP and manage configuration, card selection, and initialization. The FPGA's and microprocessor's tasks in the pipeline are distinctive enough that they can be kept separate in most situations. In the overall pipeline arrangement, the FPGA is generally "closest" to the RF front-end, being the first to affect receptions. It reads in and writes out to the USRP and PC as needed, using the device nodes that the drivers have established. The GNU Radio portion takes in the data stream from the output device-node of the FPGA, and interprets it as a file source.

From here GNU Radio can instruct the microprocessor to do any data management required, and then output the received data in whatever medium is desirable (sound, video, another file, etc.). For transmissions, the reverse of this process is true. GNU Radio will manage data from the

source device (microphone, camera, etc.) in preparation for transmission, and deliver it to the FPGA as though the FPGA were a sink. The FPGA will do its modulation tasks, and stream the modulated signal to the USRP (using GNU Radio, MATLAB, etc.) for transmission.

The FPGA really shines at implementing specific tasks in hardware. The computer is ideal at moving data around and manipulating it. Our scheme uses each component for its ideal task. In addition to providing an interface, device nodes allow for GNU Radio to be replaced by any software that can stream to/from a file, such as MATLAB. The reason GNU Radio is mentioned by default is that it is beneficial for the RF front-end we are working with (the USRP). Since the RF front-end itself is variable with respect to our FPGA, this platform is not limited to GNU Radio's standard API.

4.2 An Exercise – FM Transmission with GNU Radio

In initial attempts at FM radio manipulation, there were some technical difficulties. Investigation into the problems entailed rereading the GNU Radio tutorials by Dawei Shen, as well as consulting with grad students in the lab [19] [20]. The tutorials helped to narrow the problem down to being one regarding the configuration of the daughterboards on the two USRPs. With the provided programs `usrp_siggen.py` and `usrp_oscope.py`, a signal could be generated and transmitted from one of the USRPs, and a graphical oscilloscope could be used to verify that the signal was being received. After a couple of earlier failed attempts at simply running the transmission and reception with file storage (resulting in static as before), using this graphical diagnostic swiftly and conveniently verified which motherboard configuration functioned properly. This entailed connecting RX_A of one USRP's Basic_RX daughterboard and TX_A of

another's Basic_TX daughterboard with a copper wire. As is visible in figure 5, the configuration described functions properly, delivering the signal to the receiving computer.

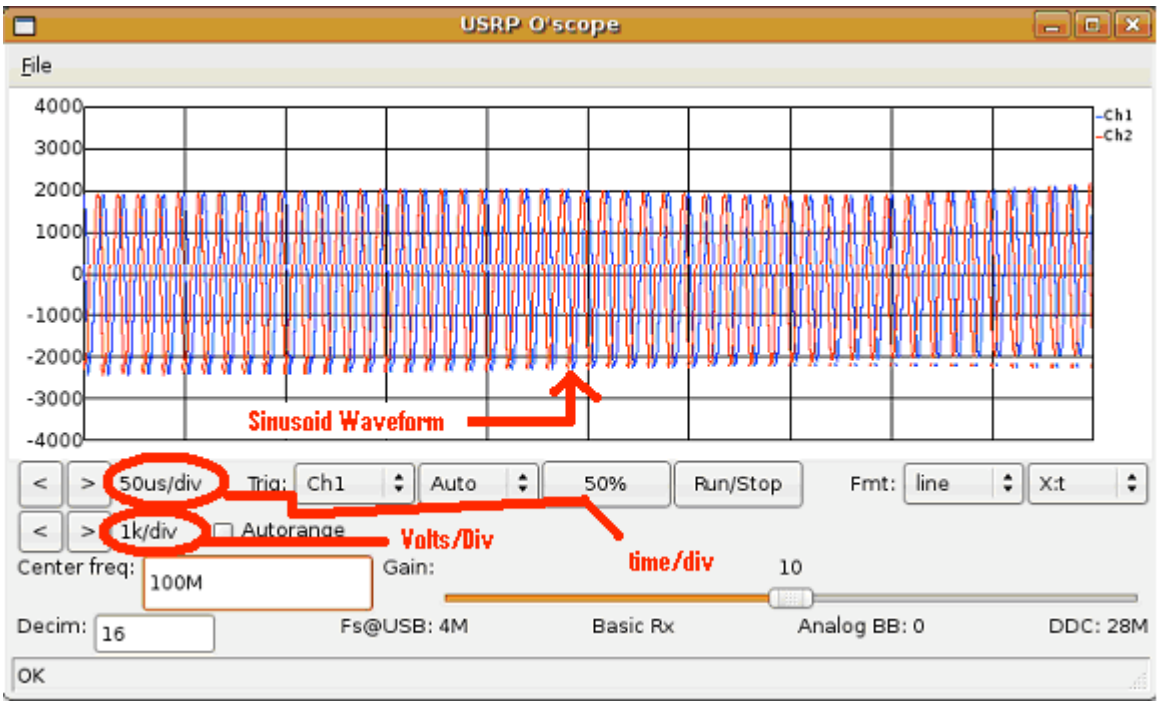

Figure 5: Daughterboard Testing Signal. As can be seen, the signal output is a sinusoid, with amplitude at approximately 2000. The center frequency upon which the signal is received is 100 MHz.

With that obstacle set aside, the efforts that were taking significant time now moved swiftly. However, there were still some slight modifications made, which made the receiving end easier to configure, as well as confirm to be functioning. Namely, this involved modifying `usrp_wfm_rcv.py` instead of `usrp_wfm_rcv_no_gui.py` for receiving the transmission and writing it to a file. Initially the non-GUI version was used as a base since the code was less complex, but the GUI version bore a greater resemblance to the code in the tutorial. It also displayed the current frequency and volume settings, as well as pre- and post-modulation signals (see figure 6). With this information, it was possible to adjust the gain and volume settings, and

observe the effect. Ultimately, a gain of 50 dB, a volume setting of 100, and a center frequency of 100 MHz were used to provide a file with a suitable tone. Some of the volume was still lost from the receiving end, but this could have been resolved by increasing the gain further, or simply using a tone with greater amplitude such that the loss was less noticeable. However, it was sufficient to conclude that the tone file was indeed being transmitted from PC to PC. The tone was noticeable when played using `audio_play.py`.

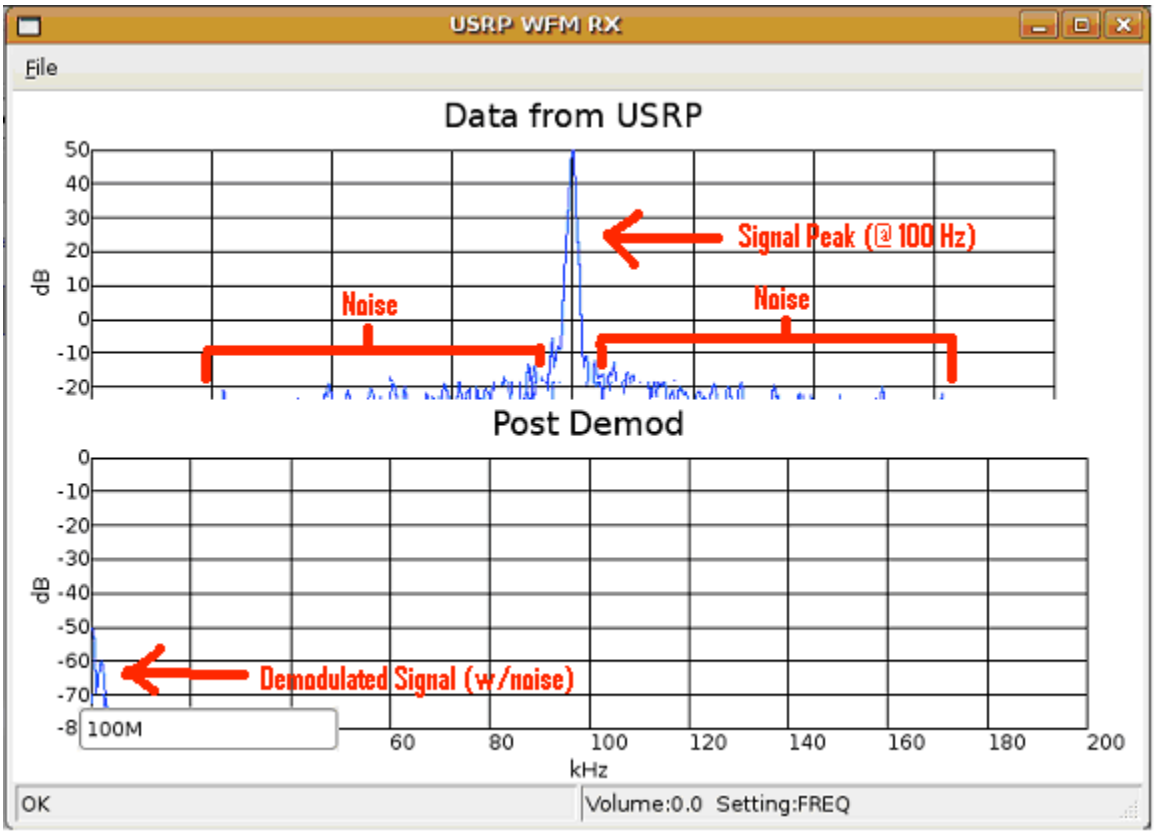

Figure 6: Frequency response of FM Signal Reception. Prior to demodulation, there is a distinct peak at 100 MHz, with smaller peaks and noise trailing off. Post-demodulation, the frequency response is what figure 3 indicates: a sinusoid at rather low frequency (approx. 500 Hz).

4.3 Preparing FM Transmission applications for FPGA Integration

The integration of the FPGA into the receiving computer, with GNU Radio in a pipeline took priority. Whether the FPGA was integrated into transmission or reception was an arbitrary decision, though it does determine which direction program implementation initially progresses in.

To this end, most of the PC-side resources were already available or almost available at that point. The components required were as follows:

- An FM transmission program that takes data as audio from any specified file
- An FM receiving program that receives from a file (ultimately the device-node for FPGA output), demodulates, and plays back or stores as another file
- Programs to generate/play back audio files (`audio_to_file` and `audio_play`, already provided in gnuradio-examples).

The FM transmission program was hard-coded to accept files with specific names (`audio-N.dat`, where N is a number from 0-7) in the same directory as the program. With some modification to this, the program was made such that the name and location of the file to be transmitted could be specified in the command line. To confirm functionality, the same test that was implemented with the hard-coded file source was run again, but giving the program a file with a different name and location. The results were the same as before – the tone was received on the other computer, with volume slightly reduced. This program will prove useful in a wider

array of situations, going beyond testing – ultimately the user would want to transmit files that were not necessarily in the same directory as the program.

The FM reception program receives data from a specified file (i.e. the device-node), demodulates it as though it was FM signal from a USRP, and stores it away in another file. Given prior experience modifying sources and sinks, and testing the reception code with a USRP source, the program is somewhat reliable. However, the program needed FPGA loopback implemented before we could be certain it was functional. When the FPGA was ready and set up on a computer with a USRP and GNU Radio, all that needed to be done to the program in addition was specifying the device-node directory when executing the reception program in the command line.

To prepare for the successful integration of the FPGA, one question still remained to be answered at that point – does GNU Radio treat device-nodes as ordinary files? To ensure that the device-node concept functioned properly with GNU Radio, a quick test of the functionality of a preexisting device node, `/dev/urandom`, was executed using `audio_play`. As expected for being supplied random integers, the program output consisted of distinct, loud white noise. So, it can be said with that in cases requiring a file to be specified, a device-node can be specified in its place.

Either way, the pipeline can be implemented in its simplest form on the software end at this stage. With FPGA drivers that are up to speed, all that needs to be known in addition is the

directory and name of the FPGA output device-node. The primary GNU Radio milestone for the first term was achieved with the completion of this pipeline.

4.4 The GNU Radio Signal Processing Block

At the beginning of B term, the development of the signal processing block has hit a couple of obstacles. First, there were many errors that needed to be worked out in the automake configuration of the block. These were largely due to the abstract nature of the automake system, and the fact that the tutorial was outdated and vague in that section. The only really way to ensure the automake functioned properly was to attempt compiling and see all the errors that came up. There were many such errors, most of which lay in the files `configure.ac` (Error 1) and `randsig.i` (Error 2, Error 4). These were relatively minor syntax errors in retrospect, but were difficult to spot because they were either small or did not appear as issues in the tutorial (which was written using an older version of `configure.ac`. The two most time-consuming errors to deal with were errors 1 and 2.

```
/usr/bin/m4 failed with exit status: 1
' is already registered with AC_CONFIG_FILES.
../../lib/autoconf/status.m4:300: AC_CONFIG_FILES is expanded from...
configure.ac:126: the top level
autom4te-2.61: /usr/bin/m4 failed with exit status: 1
autoheader-2.61: '/usr/bin/autom4te-2.61' failed with exit status: 1
' is already registered with AC_CONFIG_FILES.
../../lib/autoconf/status.m4:300: AC_CONFIG_FILES is expanded from...
configure.ac:126: the top level
autom4te-2.61: /usr/bin/m4 failed with exit status: 1
automake-1.10: autoconf failed with exit status: 1
```

Error 1: This was an issue with the configure.ac file. It was difficult to identify, since the latest version of the file differs from the one showcased in the tutorial on how to write a signal processing block. It was addressed by swapping out the more recent version for the older version, and then pulling a fresh copy of configure.ac from howto-write-a-block and modifying it again.

```
randsig_source_ff.h: In function 'PyObject* _wrap_source_ff(PyObject*,
PyObject*)':
randsig_source_ff.h:74: error: too few arguments to function
'randsig_source_ff_sptr randsig_make_source_ff(double)'
randsig.cc:4335: error: at this point in file
```

Error 2: This error appeared at first to be an issue with the header file. However, it was actually a matter involving the file randsig.i, in which there was a parameter mismatch with respect to the rest of the code.

Ultimately, the processing block compiled and tested successfully.

Another technical problem had presented itself with the implementation of the processing block, and this is one that has yet to resolve itself. While the module can be imported in the test `*.py` file by using the line `import randsig`, using this line did not work properly in a `*.py` file placed in the GNU Radio example directory. Instead, error 3 was displayed.

```
Traceback (most recent call last):
  File "./usrp_randsiggen.py", line 4, in <module>
    import randsig
ImportError: No module named randsig
```

Error 3: "Missing" randsig module. The program could not find our new library, and did not function as a result.

This implies that the module cannot be found. Probable causes for this were investigated – changing the `*.py` file's directory such that it is the same as the one where the working `*.py` is located, changing the import line to `from gnuradio import randsig`, checking the directory where all modules should be stored (`/usr/local/lib/python2.5/site-packages`) to ensure that it is in fact located there – all of these possibilities checked out fine. The issue was resolved within a few days. It was a matter of the pythonpath being unspecified. This still seemed unusual, since other GNU Radio applications functioned properly, but specifying the path got rid of the error. However, this only revealed another error (Error 4)

```
Traceback (most recent call last):
  File "./usrp_randsiggen.py", line 122, in <module>
    main ()
  File "./usrp_randsiggen.py", line 96, in main
    fg.set_interpolator (options.interp)
  File "./usrp_randsiggen.py", line 29, in set_interpolator
    self.siggen.set_sampling_freq (self.usb_freq ())
  File "/usr/lib/python2.5/site-packages/gnuradio/randsig.py", line 100, in
set_sampling_freq
    return _randsig.randsig_source_ff_sptr_set_sampling_freq(*args)
AttributeError: 'module' object has no attribute
'randsig_source_ff_sptr_set_sampling_freq'
```

Error 4: Missing randsig_source_ff_sptr_set_sampling_freq attribute. It is a variable not defined in randsig.i that is used in the *.py implementing the processing block, which causes the program to terminate.

This immediately raised red flags about either the header file or the `*.i` file not containing information about `set_sampling_freq()`. Sure enough, the `*.i` file had *no* accessors or manipulators declared, and this was changed to attempt to remedy the error. `randsig` was rebuilt, though, and the error remained. Even with deleting all files generated by the last make and running make again, it still remained. This error message was dealt with by changing the `*.i` file as mentioned above, and deleting the copy of the `randsig` module installed in `/usr/lib/python2.5/site-packages/gnuradio` as opposed to `/usr/loca/lib/python2.5 /site-packages/gnuradio`. At this point, after editing out a couple of syntax errors that did not get detected in `usrp_randsiggen.py`, the python file executed without errors. The waveform would either change erratically, or always remain the same. However, it was producing improper waveforms on the receiving oscilloscope. Modifying the code to integrate `srand()` and ensure that the waveform object was only initialized once, proved ineffective. At this point, it was thought best to turn to establishing a working secure shell (SSH) tunnel now that one of our lab computers does not seem to be functioning properly.

4.5 Digital Communications – Implementing DBPSK

4.5.1 A Demo – DBPSK Modulation over TCP/IP

The differential binary phase-shift keying (DBPSK) tunnel is a fairly straightforward process, implemented on two computers: our workbench computer (containing the FPGA), and our primary lab computer out of two lab computers that were accessable. The primary lab computer worked as expected on the first try. However, our secondary lab computer and the workbench computer both had technical difficulties. The former seemed to have something malfunctioning within the boost C++ library that was installed, and the latter was missing the Universal tunnel/network tap (TUN/TAP) Driver. With brief collaboration to properly configure our workbench computer, we had two computers which supported `tunnel.py` (included in the GNU Radio trunk for implementing TCP/IP tunnels). DBPSK modulation can now be set up on both terminals (workbench computer's example shown below), but lack of privileges on the primary lab computer made scheduling a test for simultaneous operation take longer than expected. It was a trivial matter from there, once the proper privileges were available.

There were some minor technical difficulties while implementing tunnel. The RFX daughterboards were transmitting, but not receiving packets. This was at first resolved by switching to basic TX/RX, but that operated on lower frequencies (in the MHz range rather than GHz), and as such gave less ideal performance in terms of speed, but more importantly range. Ultimately, we realized that the fact that all other wireless on campus was operating at 2.4 GHz resulted in interference with our tunnel. With that, we attempted to operate the tunnel on RFX boards at slightly a slightly different frequency, and it functioned properly. Another way of improving the speed of throughput was to reduce the maximum transmission unit. This way, if

errors occurred in packets, smaller amounts of data would be lost and require transmitting again. The ideal number found for this application was around 300 bytes. This was determined through trial and error. 300 bytes was the highest packet size that did not result in packet loss, and anything less than this would increase transfer times unnecessarily. This may vary from situation to situation, but it can be trivially changed with `ifconfig`.

We have investigated the inner workings of `tunnel.py`, and realized that poor documentation of where certain signal processing blocks are and how they work is a tremendous drawback. This resulted in consulting the mailing list, and attempting a temporary solution while waiting for a response. That is, to write code to manipulate the simulated Ethernet connection between the two USRPs on the OS level. Initial attempts at this were highly cumbersome – first generating a signal and storing it to a file (similar to what has been done before), then using system calls to execute SCP in the shell, requiring python executables to be manually run on both ends. Working with Alex, another solution was designed. This involved compressing a file and moving it through an SSH pipe. The SSH pipe called an executable file on the receiving computer, which established a pipe in the opposite direction and sent the file back to a different directory on the transmitting computer. This could be done with manual execution of only one file (`file_transmitter.py`), and was far more effective than calling SCP.

This is still not the ideal case. It works on the OS level, exploiting SSH to accomplish the required goals, and as such is somewhat slow. It is at least a good exercise in implementing GNU Radio applications for other tasks. It would be much more efficient to redirect transmitted data with GNU Radio directly. The most significant question is how to supply the digital

transmission/reception applications (`benchmark_tx.py`, `benchmark_rx.py`, or processing blocks that are subsets are these) packets or streams of data directly.

4.5.2 Initial Attempts at a Lower-Level Approach

A data streaming design works in theory, but not in practice. This is due to a hardware limitation preventing daughterboards from transmitting and receiving simultaneously, which is a requirement for data streaming. The tone transmitted was not received, and the approach had to be discarded in favor of packet-based data transfer. However, this took a significant enough portion of time to warrant mentioning here, as a path not to go down for future researchers or as a starting point to work off of if the hardware flaw prohibiting simultaneous transmission and reception of streams is no longer an issue in the future.

Much of the time investment for this design was directed toward seeking a reference for what signal processing blocks were available to us, as well as how the `benchmark_tx` and `benchmark_rx` applications work. Ultimately, we boiled it down to the use of lower-level blocks than `benchmark_*`, and avoided using them in this case entirely. This is because they inherently include modulation blocks, and we will want to insert already modulated data into the pipeline from the FPGA at some point. As a result, we replace the benchmark transmission/ reception with a series of less complicated blocks, including `file_source()`, `file_sink()`, `dbpsk_demod()`, `usrp_source_c()`, and `usrp_sink_c()`. The arrangement of these blocks is expressed graphically in figure 7, with figures 8 and 9 showing lower-level contents of the GNU Radio flowcharts.

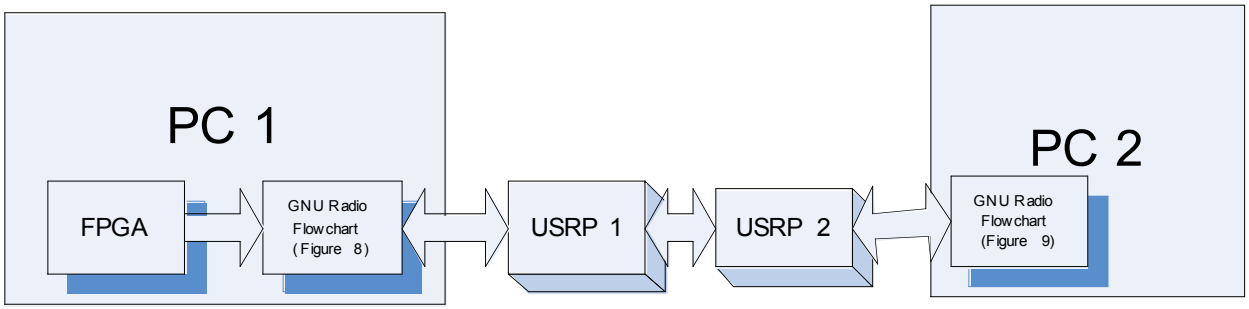

Figure 7: A top-level diagram of the pipeline. The FPGA on PC 1 supplies data to the flow graph, which relays across the USRPs to the PC 2. The flow graph of PC 2 redirects anything received back to USRP 2. The The two GNU Radio flow graphs are quite different and vary in complexity, as is show in figures 2 and 3.

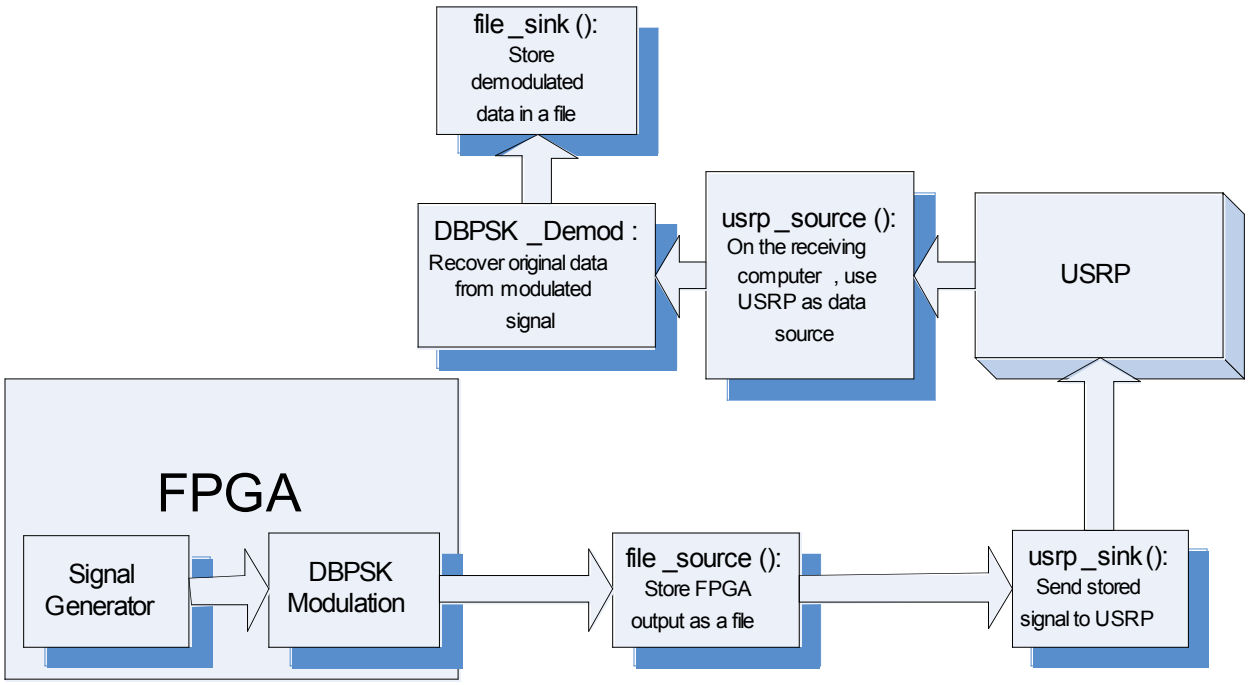

Figure 8: The planned GNU Radio flow graph for PC 1, along with the tasks allotted to the FPGA. This includes all signal processing blocks that should be needed, presuming DBPSK modulation was implemented on the FPGA. The FPGA is represented by a character device node, and treated as a file source. Anything *received* by the USRP from PC 2 is interpreted as a USRP source, demodulated as DBPSK, and stored as a file. In the initial development process, prior to writing a DBPSK modulation block for the FPGA, the signal processing blocks sig_source_c() and dbpsk_mod() can be used as a replacement. For this to function properly with only the blocks in the diagram, the FPGA must output the generated signal as complex numbers. This design was not completed, as streaming data transfer only works unidirectionally with GNU Radio.

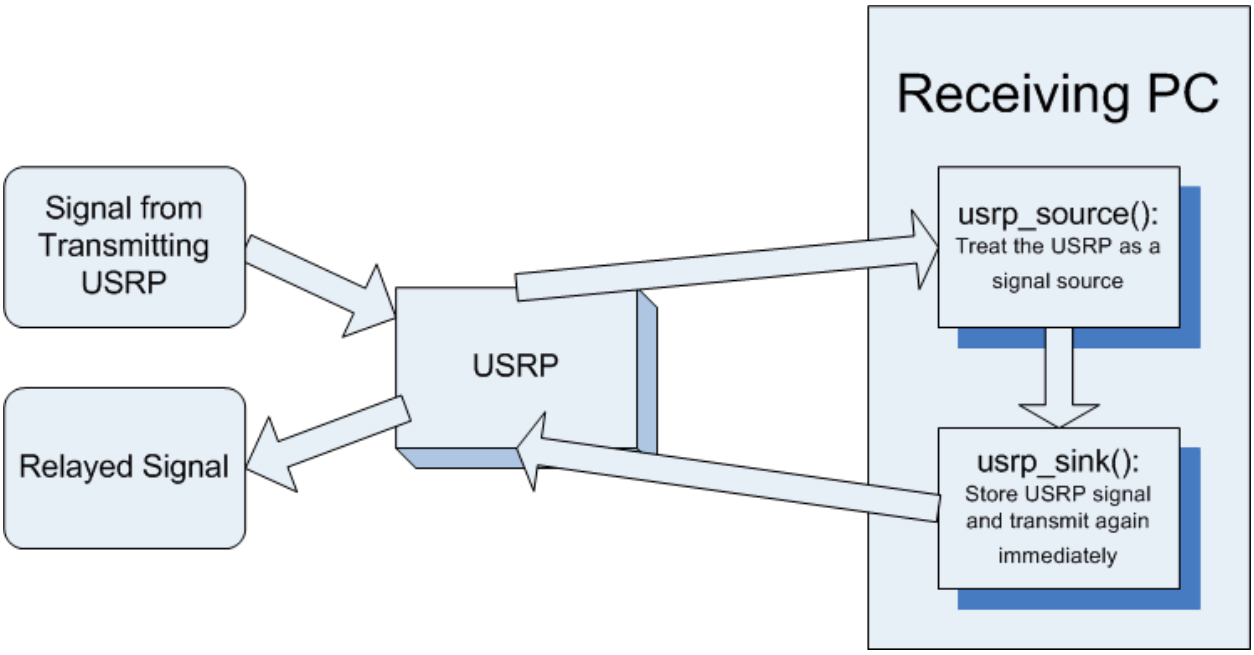

Figure 9: The GNU Radio flow graph for PC 2. Quite simply, it interprets the USRP as a source, and immediately directs any input back to the USRP as a sink. This is a loopback, so the receiver is a "dummy", for the moment only determining the functionality of the transmitter.

4.6 The Current Concept – A Packet-Based Approach

4.6.1 Unidirectional Communication

To resolve the problems present in stream-based data transfer, we further examined the `benchmark_tx.py` and `benchmark_rx.py` applications. They could be adjusted to transmit to/from files by implementing file I/O, having payload be read from a file using `read()` and the received payload (with the exception of the two least significant bits) using `write()`. It was a matter of knowing how to implement file I/O in python, and knowing where to place the read/write functions. We had come to this realization after stumbling upon similar modifications

for `benchmark_tx`, and developed the `benchmark_rx` modifications completely independently, using the newfound knowledge that we could use file I/O to modify and read the payload string.

With these modifications, we now have unidirectional file transfer done properly – in a way that can later be altered for new modulation options (the module where the benchmark files select their modulation type, `mod_pkts`, has been identified), and is far more favorable to integrating the FPGA. For instance, the function generator we used at the end of A term for FM transmission could be implemented as a file source again. And if a DBPSK block is written for the FPGA, that can be used and GNU Radio modulation can be removed.

4.6.2 Bidirectional Communication

The initial attempt at a relay for bidirectional communication (stream-based) also had to be abandoned for a packet-based application, as the RFX2400 boards inherently cannot transmit and receive at the same time. The initial relay consisted of a USRP source and sink receiving and transmitting constantly (in theory). This does not work, because transmitting takes priority and the application never receives.

To remedy this, we shifted away from creating a stream and tried instead to implement packet exchange, as we had done in the alterations of his plan for unidirectional file transfer the prior. This requires taking timing constraints into consideration, since one computer must be receiving when the other is transmitting. The transmitter has a discontinuous mode, which sends a certain number of packets before waiting for some period of time. This proved ideal for synchronization, since the relay side could be made to know to receive a certain number of packets and then

transmit, and the originating side could wait long enough after packet bursts to receive a series of packets the same as it had just transmitted.

This application runs, but is unreliable. The RX callback function in the relay application (used for both transmitting and receiving) is called whenever packets are received. Therefore, for the last packet in a burst the relay USRP is attempting to transmit and receive at the same time, so the packet is never sent back. If the number of packets in a burst is significantly large and the file is a sine wave stored as a series of floats, this causes only minimal distortion to the final file, and it is still usable. The greater problem is that many packets transmitted by the relay are not properly received, and are dropped. Effectively implementing bidirectional communication had to begin on a smaller scale, transmitting individual packets of known contents rather than large files.

Implementing transmission of single packets with known packet contents in two directions proved to be a far more straightforward and less error-prone task. Over the first weekend of C term, we wrote programs for transmitting a user-specified string in a packet, relaying the packet back to the transmitting computer, and displaying the original string contents/storing them in a file. All of these tasks are now performed reliably. This was bought together with the FPGA in loopback once the driver (see Chapter 5) was functional. For more on the loopback integration results in this example, see Section 6.1.

Chapter Summary

Over the course of this project we applied GNU Radio to a variety of tasks, and checked FPGA functionality (at least ostensibly, in the form of a loopback) at every stage. Ultimately, the FPGA

integration was intended to be used to replace certain tasks within the GNU Radio pipeline. In the following chapter, we will establish the architecture within the FPGA, and the means by which it interacts with the pipeline.

5. FPGA Architecture and GNU Radio Integration

5.1 Driver Development

The goal of the driver in this project was to integrate GNU radio and other applications such as MATLAB with the hardware running on the FPGA. Even though many environments can be extended to add new functionality using mechanisms such as MATLAB's MEX modules and GNU radio signal blocks we decided against using a specific solution similar to the solutions mentioned above due to accessibility reasons. We did not want to create a solution that tailored specifically to one data manipulation suite. Instead, we chose to implement the interface to our hardware as a UNIX device-node.

If we look at the way file-IO is implemented in a modern operating system we find a buffered bidirectional stream of data. This is why in UNIX derivitive operating systems the file metaphor is used to access a wide variety of hardware. A device driver would create and associate itself with a special virtual file called a device-node using device file system (devFS) or UNIX File System (UFS) and then read and write from the other end of the buffer. On a modern Linux system the following command:

```
cat /dev/urandom >> /dev/dsp
```

will result in static being played out of a user's speakers. The command reads the file "/dev/urandom" which is a device node created by the random number generator. This file provides random data every time it is read from. Then the output is redirected to the file

`/dev/dsp` which is the device node for the computer's sound card. Any writes to this file will cause the written value to be fed into the DAC in the computer's sound card. This forms a very rudimentary data pipline as depicted in figure 10. This example shows how the "device as a file" metaphor can be used to access hardware. A more controlled form of the above example could be implemented using the 'dd' command:

```
dd if=/dev/urandom of=/dev/dsp bs=4 count=10
```

This command would copy 10 units of four bytes of data from the random number generator to the sound card.

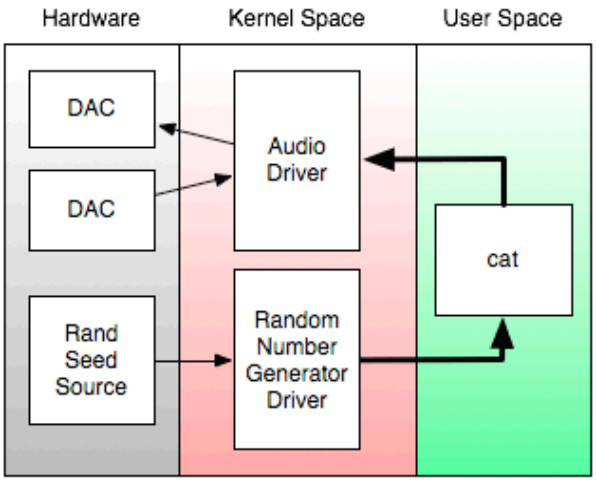

Figure 10: Visual Representation of cat /dev/urandom >> /dev/dsp forming a pipline.

For our platform, we will represent the configuration interface and both ends of each synthesized pipeline as device-nodes. Figure 11 shows how our driver would interact with the hardware on one end and the host application on the other end.

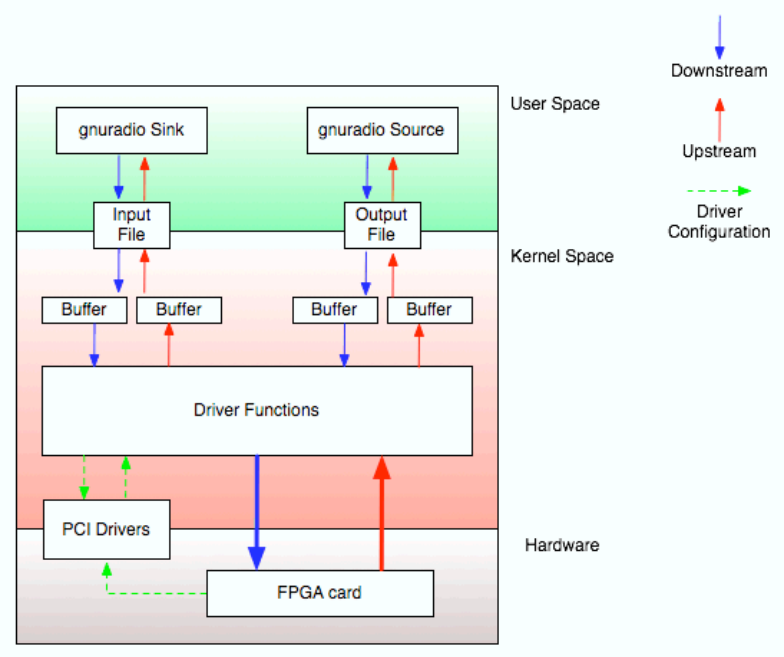

Figure 11: A flowchart of the driver we use to push data to and from the FPGA

This method of design in addition to providing a easy to use interface abstracts the hardware specific details and leaves them in the domain of the driver. As long as the same interface is provided to the software portion of this problem the hardware and driver can be modified without adversely affecting the software and breaking compatibility.

5.1.1 PCI Initialization

PCI express is an extension of the original PCI bus that replaces the physical parallel bus with a much faster multi-serial packet based PHY. From a software level, however, it was designed to look and behave similar to a PCI bus so much so that a system that is unaware of PCI express can treat it as a plain old PCI bus and function properly.

PCI was designed to replace industry standard arhitecture (ISA) which was an extension of the 80286's processor bus. In addition to increasing performance, PCI included features such as architecture independence, plug & play, auto configuration, and hot plug support. Both buses allow the host system to communicate with hardware using memory mapped IO. This means that once our card is configured we can access it as if it was memory. However, some of these advanced features make setting up the card complicated.

When a PCI card powers up initially, it only responds to configuration requests. On a PCI bus there are three memory spaces: Memory, IO, and configuration. Memory and IO address spaces behave as they do on ISA, cards get mapped to specific address ranges and can be accessed by using IO and memory transactions. However, the configuration address space exploits geographic addressing. This means that no two cards will have conflicting configuration addresses. Using this configuration space the host can then configure specifically what addresses the card can respond to and achieve plug and play support without having the IO and memory be geographically addressed.

When the system boots, the BIOS (or the OS) crawls through the bus and makes sure that no two cards are conflicting with each other and have their memory regions (specified in the configuration registers) mapped into system memory. Our driver has to associate itself with the card and perform some configuration. The card type can be identified by the vendor and device pair. XILINX has a vendor ID of 0x10EE and we assigned the device ID of 5050 to our card. We then register two callback functions with the kernel telling it what it should do if our card

(specified by the tuple) is added or removed from the system using the pci_register_driver() function.

```
static struct pci_device_id  pci_device_id_DevicePCI[] =
{
  {VENDOR_ID, PCI_ANY_ID, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0},
  {}
}; // PCI identification struct. Associate with any xilinx products.

struct pci_driver  pci_driver_DevicePCI =
{
  name: "MQP_TEST_VIRTEX5",
  id_table: pci_device_id_DevicePCI,
  probe: device_probe,  // pointer to "card added" function
  remove: device_remove // pointer to "card removed" function
}; // describe our driver to the kernel.

res = pci_register_driver(&pci_driver_DevicePCI); // register functions
      if (res==0){ // driver sucsesfully loaded
            printk("PCIe driver loaded!\n");
      } else { // failure. Return an error.
            printk("something went wrong!\n");
            return res;
      }
```

Using the callback "probe" mechanism, we now have a way of finding and working with our card. The "Add" function has to do three things. Initialize the card, Find out what base address has been assigned to the desired BAR (memory block) and configure the system's memory management hardware to set up a mapping between our virtual memory and the system's physical memory for that specific range. The equivalent "virtual base address" is stored in a struct for later use.

```
// enable card
ret = pci_enable_device(dev);
  if (ret < 0) return ret;
printk("Device enabled sucsesfully\n");

// configure translation for BAR0
mem_start = pci_resource_start(dev,0); // reigon zero. BAR0
mem_len = pci_resource_len(dev,0);
map=request_mem_region(mem_start,mem_len,"driver");
ml506.vmemadd=ioremap(mem_start,mem_len); // save the virtual address
ml506.memadd=pci_resource_start(dev,0);
ml506.memlen=pci_resource_len(dev,0); // We now know where our card is in
memory and can access it using memory transactions.
```

The “remove” function removes this mapping and disables the card, cleaning up after the “add” function. Once the “add” callback function is called by the kernel our card can now be accessed by simply writing into the memory range specified by the “virtual” base address and the length of the memory block.

5.1.2 Character Device Initialization

A character device from the user’s point of view is a file. One can read and write to this file as well as seek to an arbitrary position in the file much as you do with any other file. In this case, however, all file operations are handled by our own functions that read and write to registers on the peripheral card.

There are two numbers that represent our character device when combined. These are the major number assigned to the driver and a minor number that the driver assigns to the character device. In our driver since we only have one device we chose a minor number of zero. To register a character device we first need to create a structure with pointers to our handler functions. We then pass this pointer to the kernel along with a minor number using the `register_chrdev()` function. This function returns our major number. To create the device node we use the command mknod this way:

```
mknod /dev/device 253 0
```

This makes a character device and connects it to driver 253 char device 0. Any interactions with this file will cause our callbacks to be called.

5.1.3 Character Device Data Functions

The four required character device functions are device_read, device_write, device_open, and device_release. These functions get called when our device node is read from, written to, opened, or closed. These are the function prototypes for the four character device functions:

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
```

In our situation the two important functions that contain the bulk of our code are the "`device_read`" and "`device_write`" functions. These functions contain a pointer to a user space buffer, the length of bytes to be written, and the offset into the file. For our application we ignore the offset because our file has infinite length and seeking is irrelevant in that context. The function's return argument is how many bytes where written.

The FIFO on the FPGA can only handle 32 bit integers so we need to have four or more bytes in the buffer. Our code checks the length and returns zero bytes if it is less than four. The user space library should check this return value and keep trying until it was able to write all the data. In a stream this is acceptable because data would be coming in four byte chunks. Once we have enough data in the user space buffer we need to copy it into a kernel space buffer because kernel space code and user space code have different virtual address spaces. This can be achieved with the "`copy_to_user(src,dest,count)`" function.

In our driver we copy the four bytes into an integer, change the ordering to little endian, and then write it to the FIFO input register on the card. For reads the process is identical but reversed. We wait until four bytes have been requested, read four bytes, change endianness, and copy to the user space buffer.

Our driver implements a form of flow control. Since the FIFO on the card can only hold 1024 samples we keep a counter that gets incremented on a write and decremented on a read. If a read occurs when the buffer is full we employ blocking IO where instead of returning “0 bytes read” we put the process to sleep and wake it up when data has been written to the card. The same thing happens when a write to a full buffer occurs. The downside of this method is that the read and write pipelines have to be in separate threads.

5.1.4 Driver Debugging

One of the main problems we ran into when developing tie driver was figuring out how to configure the mapping between physical and system memory. The card’s virtual address is different from its physical address because it has been mapped into the processes memory space. Initially we where not performing this translation and where trying to access the registers returned from “pci_resource_start()“ when we should have been getting them from the pointer returned by ioremap().

5.2 PCIe Interface Development Initial Attempt

We initially set 6 milestones for the development of the interface leading up to dynamic reconfiguration. These milestones are as follows.

1. Hardware test using XILINX’s getting PCIe started guide.

2. Synthesis of the provided Verilog code in an ISE project

3. Syntheses of a PCIe core without any application logic but with valid configuration information.

4. Implementation of a read only memory with a predefined pattern.

5. Implementation of some sort of stream based data transport mechanisim

6. Extension of the step 5 state machine to write to the ICAP port and to a pipline.

5.2.1 Hardware and Environment Testing Using Reference Bitmap

When we initially began development in A term the first problem we ran into was intellectual property issues. This was initially mostly due to being unaware that the trial version of the XILINX ISE software did not have all of the side applications and privileges we needed - namely, COREgen and root access. Upon realizing this, we attempted synthesize the desired core on a network PC, after acquiring and installing the proper intellectual property permissions.

However, this proved unsuccessful due to the lack of administrator privileges on ECENET, combined with XILINX having difficulties reading information from the network drive. Initially, an attempt at synthesizing the core resulted in an error message in a window as follows:

```
XST has returned an error: ERROR:Xst:2367 - Unable to create .lso file
"endpint_blk_plus_v1_7_pcie_blk_plus_gen_1.lso"
```

The licencing error returned by COREGEN when we tried to install a trial licence for the PCIe Primative. Our later solution used components that required no licences.

We assumed that this was some sort of permissions error being thrown by xilinx because the software was designed to run with administrator privalages.

We attempted saving our project on the Desktop rather than the M:/ drive. This got rid of that error message during synthesis. However, error messages regarding the IP certificate (`"Parsing of check licence val <> failed."`, etc.) appeared, and synthesis was unsuccessful. This is likely due to the fact that the IP certificate had to be stored on the same drive as XILINX in order to work properly, and XILINX is unable to read it off of the network drive (the only drive we have access to on an ECE Department PC). We believe the solution to this is to be provided the full version of XILINX ISE, to install on one of our own computers. This gives us administrator privileges on the hard drive of the PC used, rather than relying on the problematic network drive.

We were reluctant to use our provided workstation for syntheses as well as testing because whe where worried about bad hardware crashing the computer so we used a personal machine to synthesize and transferred the bit streams to our workbench computer for testing. This proved to be beneficial because initially the workbench computer would lock up as we debugged the hardware. The example provided by XILINX was an implementation of a “memory card” which would allow the host system to read and write to block-ram through the PCIe interface.

5.2.2 Compiling the Provided Example Code in ISE

Since the tutorial started from a partially generated bitmap and used batch scripts to synthesize and implement the bitmap it was never intended to be more then an a example. In order to work on the project further we would need to synthesize, from scratch, a valid bit stream from the provided UCF and Verilog files in an ISE project. Since we know the hardware development language (HDL) and UCF files provided with the example are correct and the board works the only variable in test 2 would be the development environment. We ran into problems in this stage. The first problem was with compile time directives not working properly. After reading

through the code and determining the function of the compile time directives we modified some of the Verilog such that they where not required. After this, we got the HDL to properly synthesize and implement. Even though the synthesis process was successful, the generated core would not work. Initially we thought it was a problem with IAR and proceeded to try and debug the problem but could not find out what was wrong. After searching through the XILINX community forums we finally found a person with a similar problem. In that case, the UCF file was wrong and the wrong transceiver was selected in the UCF file. After consulting the documentation the selected transceiver "GTP_DUAL_X0Y3" was connected to one of the SATA connectors. The correct transceiver was "GTP_DUAL_X0Y1"

In addition to this the pins for the differential clock where set to F3 and F4 when they where really connected to AF3 and AF4. After these changes where complete the generated bitmap worked properly.

5.2.3 Generating a Bitmap with Only the PCIe MAC/PHY

In the PCI spec the card implements 3 address spaces: Configuration, Memory, and IO. The only memory space the system itself writes to during the enumeration process is the configuration space. The system uses a specified driver, if available, to further access the Memory and IO spaces of the card because the details on how to access these is device specific and is the domain of the driver. Since no driver was loaded by default for the vendor and device ID that where programmed into the PCIe physical layer (PHY) the device should still be properly enumerated. The purpose of this test was to find the boundary between the HDL needed to properly set up the device and the application logic. Since there is no need to reinvent the wheel and write a top block from scratch when it would be identical to the one provided in the example (only with

more debugging) we decided to find the application specific VHDL and remove it in order to find this boundary.

The top block contained two instantiated modules. One was the IP core and the other was the module containing all of the application code. The top block contained a net for every one of the approximately 200 inputs and outputs of the PCIe core and tied them to approx. 200 corresponding inputs on the application logic. After digging into the application logic block we found a large portion of thoes signals, mostly configuration signals tied to hard coded values. The rest of them went into a block labeled PIO. After digging deeper the PIO module whose inputs where the two 64 bit busses and other bus control signals used to send and receive packets we determined that this was the module to remove.

After removing this module and synthesizing and implementing the project we generated another bitmap. After copying this to the FPGA and rebooting the computer we saw that the card had successfully been enumerated.

5.2.4 Addressing Throughput Issues in the Project

Due to the type of bus interface we are using some sort of buffer in hardware is a must. Both PCIe and GIGe are packet based protocols with rather large payloads and the efficiency drops below 10% when single words of data are being sent. In addition to this it is hard to guarantee arrival of the data periodically. However, the bandwidth of the bus is much higher then the bandwidth we are likely to require so having a buffer of sufficient size would allow us to ensure that there would be a constant stream of data available to the pipeline.

Initially, Alex was planning a half-FIFO (see figure 12) rather than full FIFO arrangement for the FPGA. The reasoning behind the half FIFO consisted of two main parts. The first part was an interface to the PHY that had the highest probability of working and would be the easiest to debug. The second part is to provide an interface with higher throughput and less overhead.

This design is called the half-FIFO because unlike a normal FIFO, which has a state machine for both the read and write pointers, the half FIFO only has a state machine for the portion facing the pipeline. The portion of the FIFO facing the PCIe PHY and the system is a standard memory interface. This means that the entire buffer is mapped into the host systems memory.

The previous design idea had the output of a FIFO mapped into the system's memory as a single register and subsequent reads and writes to this register would push and pull elements out of the FIFO. This is inefficient because each PCIe packet carries approximately 16-24 bytes in overhead with 4 bytes of data. A more ideal solution would be to use bulk reads and writes, which can carry up to 4k bytes of data. Managing half of the buffer in the driver allows for the use of bulk data transfer which increases the throughput by decreasing the overhead.

The driver would keep a write pointer in a register that can be accessed from the FPGA. The driver would write chunks of data at a time into the buffer and then add the amount to the write pointer. Since the write pointer would essentially be in shared memory. The state machine would be able to access this pointer and then assert control signals telling the pipeline that the buffer has filled or emptied. In addition to being a more optimal solution it is also easier to implement because blocks of memory are easy to interface with the core.

A problem encountered while implementing the FIFO was that XILINX was not generating a proper bit stream by using old files to generate a bitstream instead of newly synthesized files, even after "clean project directory" was run and all IP cores where re-synthed, and replaced several times. It would still generate an incorrect bitstream. The solution (after debugging the code, then debugging the project.) was to start a new project and import all the source files and re-generate the IP cores.

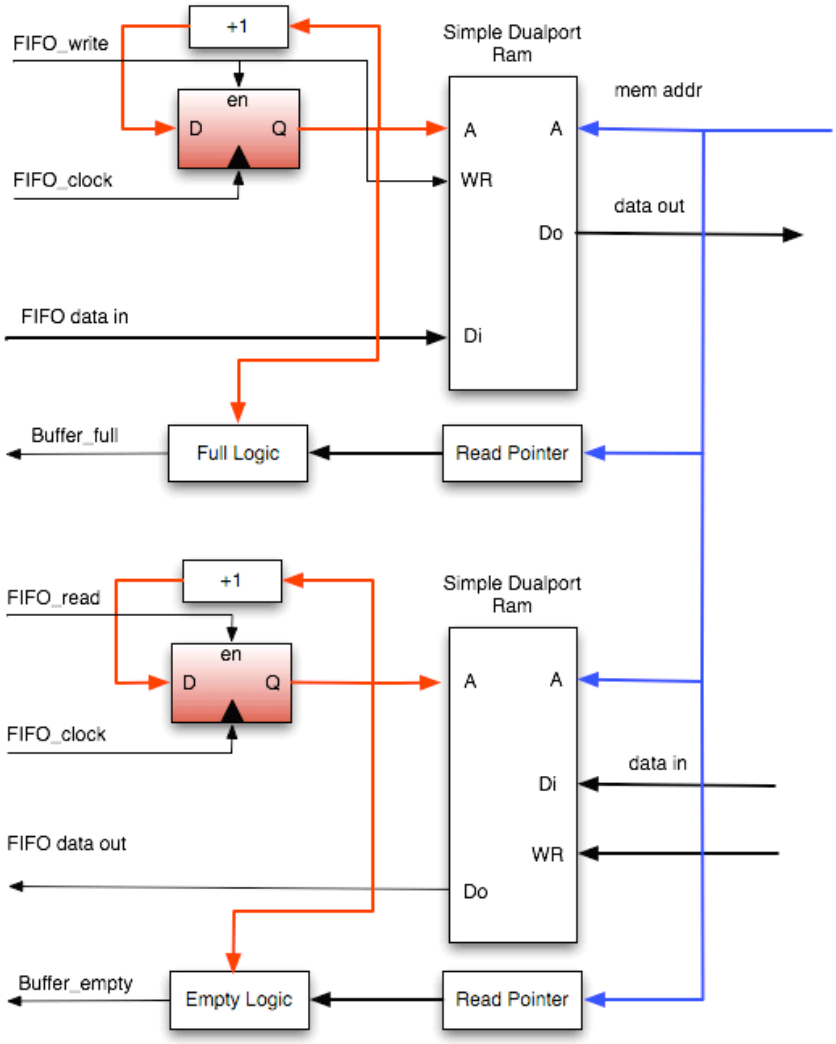

Figure 12: Half-FIFO Diagram showing our initial implementation of hardwate to pass data to and from the host computer.

The half-fifo was intended to allow the PCIe PHY to map the fifo pointers and memory into the system's address space so the driver could use a bulk memory write to update the FIFO increasing efficienty. It behaved similar to a regular fifo but the read and write flags where set by the host. Using these flags the driver running on the host could control the flow of samples through the pipline

Even though we got the FIFO hardware working in a loopback mode with the outgoing and incoming channels connected we ended up rethinking the way the buffer would be handled. Initially, the reason we picked a memory-like interface for a portion of the fifo was to facilitate bulk memory reads and writes provided that the PCIe paket decoding was updated as well. When we switched to an EDK project we decided to use the fast simplex link to PLB bus interface in conjunction with a MicroBlaze (or DMA controller). The data would arrive in bulk over the PCIe interface and be stored in a SRAM buffer (the driver writes chunk of data to card's address), then the micro blaze or a DMA controller would copy the data out of SRAM and out over the FSL interface. A FSL link itself is a FIFO of up to 1KB length. The FIFO in the FSL link in conjunction with the SRAM is sufficiently pipelined to provide a continuous stream of data.

Interfacing to the ICAP will be handled in a similar manner except the MicroBlaze will handle copying configuration information out of SRAM and into the ICAP interface while handling synchronization and other minutiae of downloading a bitmap over the ICAP interface. This ends up being a more efficient then having the driver handle all aspects of configuration due to the advantages of sending data in bulk over PCIe.

5.2.5 The ICAP Interface

The group decided to implement a more complex method of dynamic reconfiguration involving a microcontroller. The reasoning behind having a microcontroller in the system would make ICAP easier to troubleshoot because we could have easier access to the ICAP primitive as well as allow us to make the programming process more efficient and intelligent. The microcontroller and the EDK where utilized to implement this programming interface. This portion of the project was not completed tue to time restrictions.

5.3 PCI Re-implementation Using EDK

The embedded design kit (EDK) had additional modules that made it more worthwhile to replace the work we had currently done with the EDK IP then to debug it. This proved to be a quicker way of implementing PCIe due to available components such as the PCIe bridge and the PLB to FSL bridge.

5.3.1 The PCIe to PLB Bridge and Its Role in the System

The most important component of the EDK project is the PCIe to PLB bridge. This bridge, which uses the primitive as one of its subcomponents, translates memory reads and writes made to the card's BAR (Base Address Register) memory space and generates equivalent bus transactions on the EDK SOC's PLB. This bridge supports the reception of bulk packets increasing efficiency.

The role of this component in our system is to facilitate the transfer of information from the realm of the computer with the help of a device driver to the relm of the EDK project. The EDK

SOC's bus serves as an intermediary as shown in figure 13 allowing us to then manipulate this information and route it. For example, ICAP configuration information may be pushed to the ICAP interface initially or cached in SRAM while the micro blaze handles synchronization and transfer of information. The same applies to samples. Sending samples in bulk, caching them in SRAM and then having a DMA controller or the micro blaze copy them into the FSL link is more efficient then accessing the FSL hardware using the bridge alone.

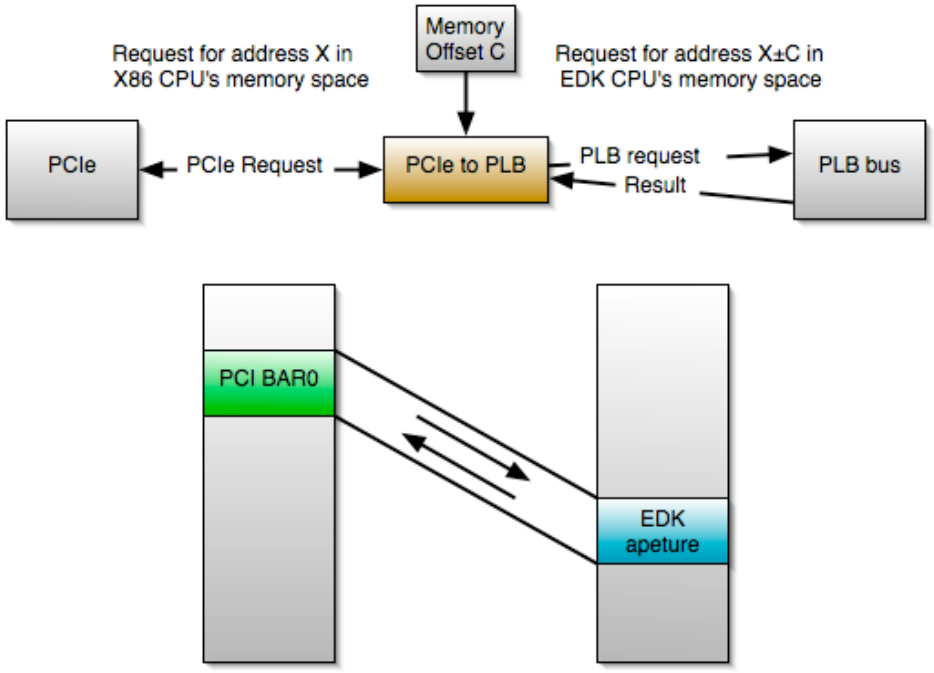

Figure 13: The operation of the PCIe to PLB bridge.

5.3.2 The FSL Link and Interfacing to the Pipline.

Getting information out of an EDK system is awkward, The only real options are to write your own PLB peripheral and associate it with the proper metadata to get the EDK to recognize it and allow you to use it or use a pre-existing component that implements an interface that would be appropriate for your module and pull that interface out of the EDK project.

For our design there were two types of external access we wanted to perform: Module configuration and data flow. For module configuration we chose to use a LMB Memory controller which connects devices that implement the same interface as a block ram to the PLB bus. We intended to use geographical addressing for each of the modules splitting up the four kbit memory space into 1000 kbit chunks by having the upper 2 address lines control enable signals to the memory interface of each module. This would give the modules 1k of memory that could be accessed by the computer through the PCIe Bridge. This memory space could be used to set coefficients and module behavior.

For the data flow, we decided that modules would implement a fast simplex link protocol. The FSL protocol is a point to point uni-directional link between a master device and a slave device. It is a 32 bit synchronous bus with flow control signals. Each module would implement a slave (input) and a master (output) interface. The PLB to FSL controller would also implement a master interface for outputting samples and a slave interface for the return data. Modules could be chained together end to end with the final link returning the data to the controller. The clock speed of the interface is the clock speed at which the PLB bus runs. An example of this configuration is shown in figure 14. In this case, clock speed is 125 MHz.

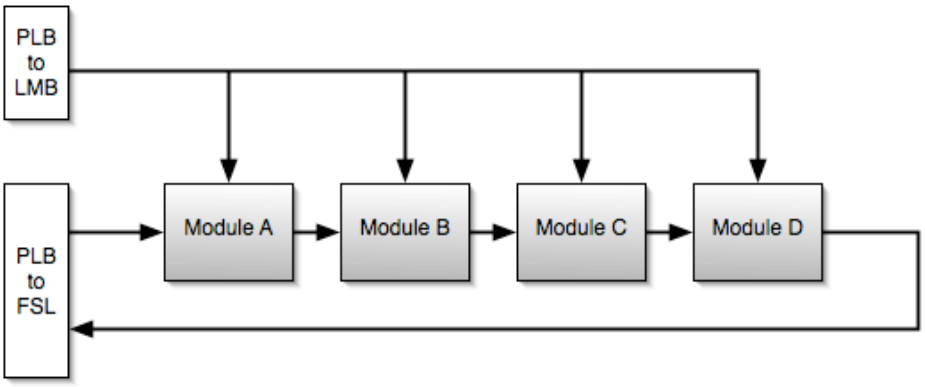

Figure 14: A full pipeline of 4 modules connected end to end using FSL links and configured using a LMB bus.

The implementation for our final demo only consists of one module and lacks a LMB interface this is mostly due to time restrictions. In the demo the group used fixed coefficients, however, a more refined version would have allowed for coefficients to be re-written using registers accessed through the LMB interface. This would have shown how we pass module configuration information to each module in addition to passing data through the pipline.

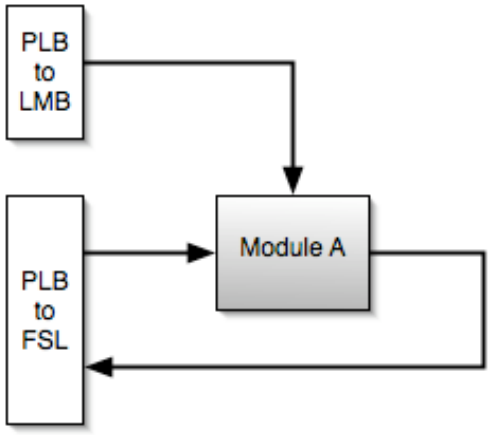

Figure 15: A simplified pipline with only one modual and a configuration interface. Our final demo used fixed co-eficients and did not use a configuration interface although future revisions of the FIR filter would allow the user to set these coefficients himself.

5.3.3 Test Environments

Initially, we would test the bitmap with XMD shown in figure 16 a XILINX debugging tool. This debugger could generate bus transactions that where identical to the transactions generated

by the PCIe bridge. It can be used immedietly after a reflash without a reboot so it was used when initially debugging the module.

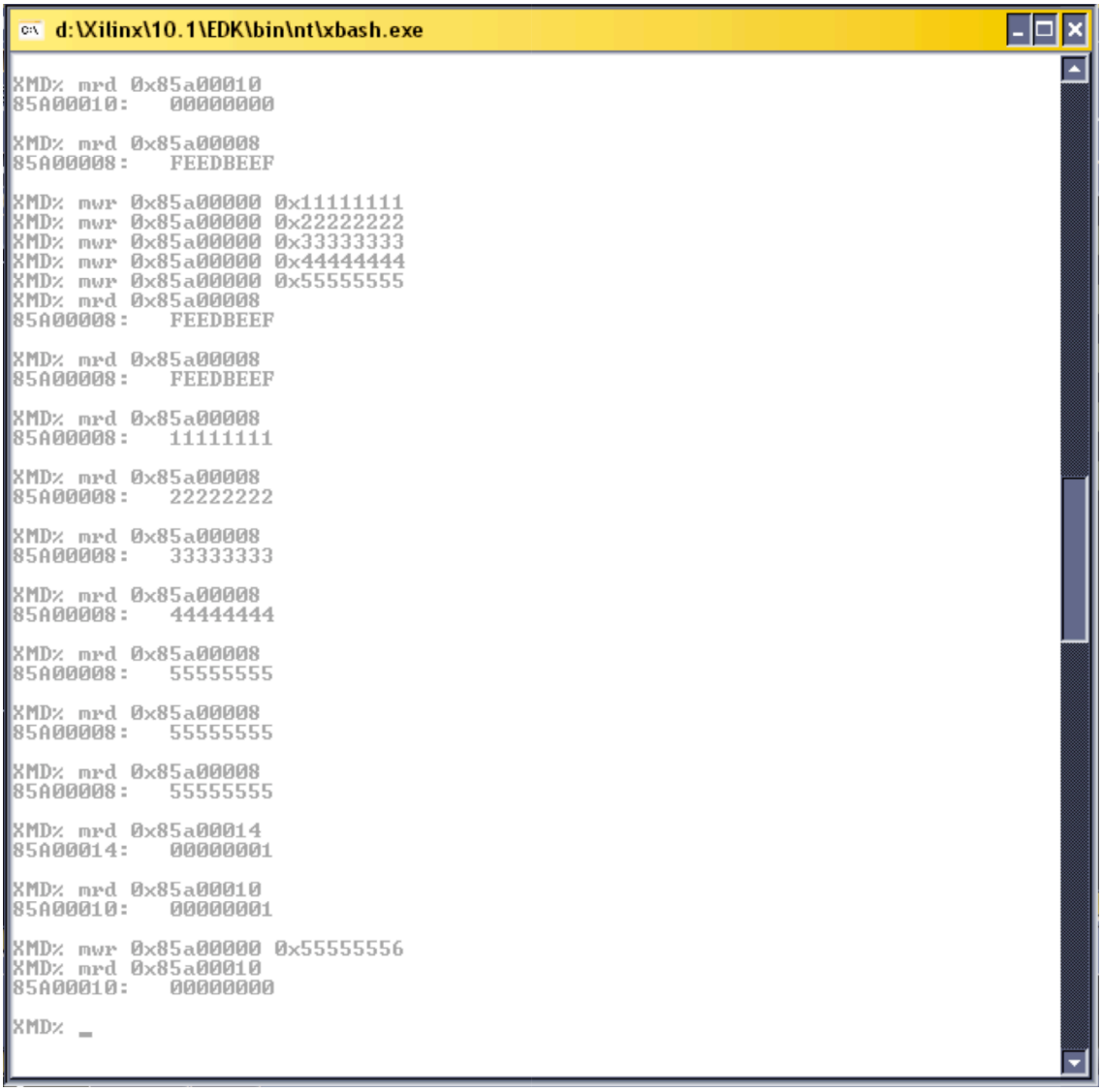

Figure 16: Verifying the functionality of the loopback interface via XMD by writing samples into one end of the FIFO and reading them from the other. The FIFO addresses for input and output in this figure are 0x85a00000 and 0x85a00008. Functionality was verified by comparing the information coming out of the link to what was sent in.

The next tool that was used was PCI tree shown in figure 17. This program allows one to bring up a card and read and write to its BAR regions without a driver. This was used to test the PCIe interface after initial testing with XDB had prooben that the filter was functional.

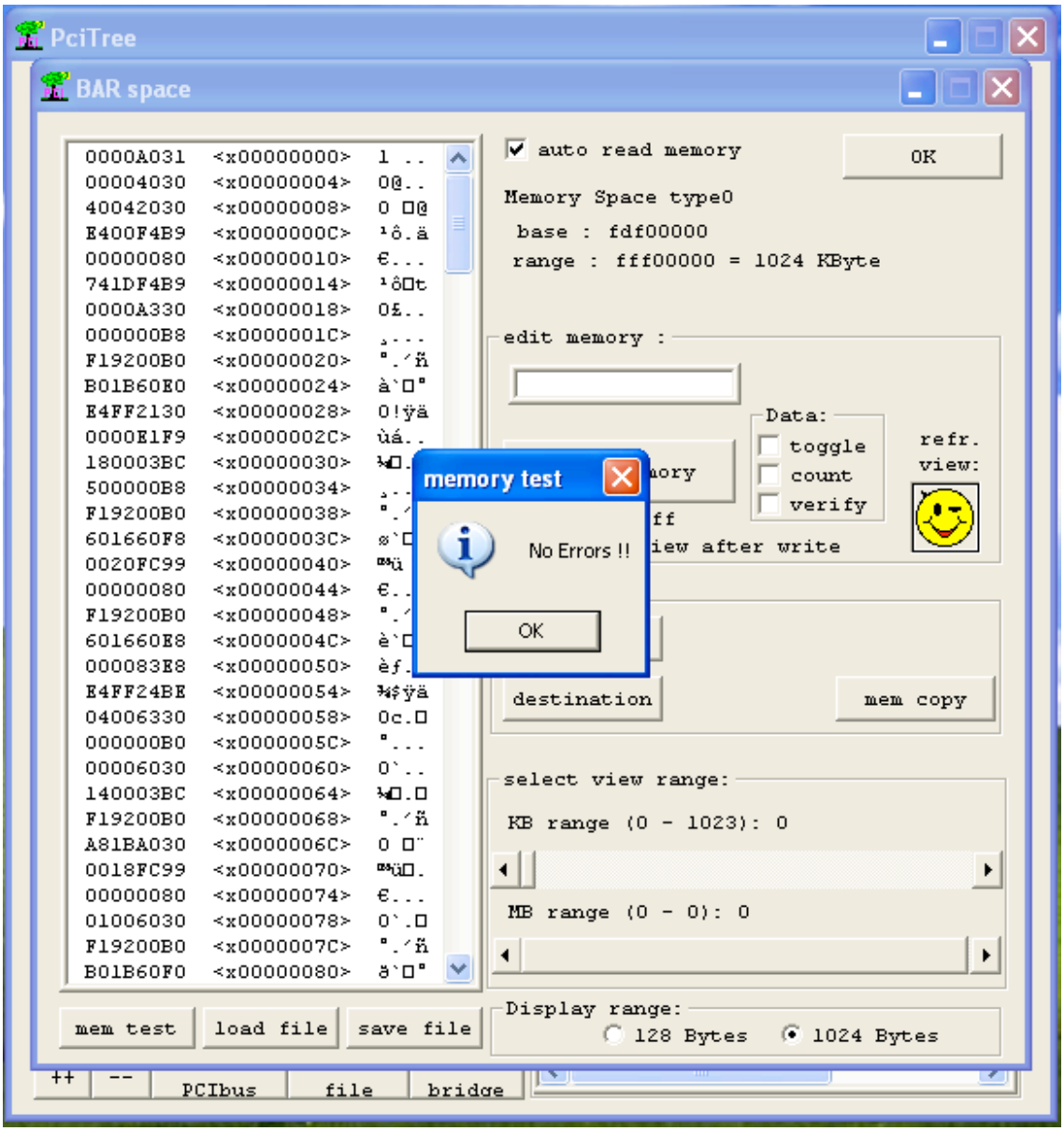

Figure 17: PCItree doing a test on the FPGA's sram via the bus bridge. This software was used to verify the functionality of the PCIe bridge.

5.4 Platform Efficiency

5.4.1 Driver Efficiency

There is currently a significant amount of overhead incurred by the implementation of the driver. Every time a sample is read or written to the card results in one syscall and a transition into kernel space during streaming this overhead doubles. In addition to this performance is also lost during the context switch from the reading and writing processes being swapped out. Ideally, Sending code to the driver in bulk and then copying it to the card sample by sample from kernel space would increase efficiency. Restructuring the writing (and reading) to more resemble the pseudo code would accomplish this goal. The inefficient method was chosen for ease of implementation:

```
While(counter>=4){
      Write32(address,*(buffer++));
      Counter = counter-4;
}
      return counter;
```

5.4.2 PCIe Transfer Efficiency

Due to the way we transfer data over the PCIe bus we incur a 83% performance hit. This is due to the ratio of data to header information in the packet. Every packet requires 5DW (32 bit Data Words) of header. Since we are only sending one 32 bit sample at a time the stream of information is 5/6 header and 1/6 data. The bus was designed to transfer data in bulk and if the maximum TLP payload size of 4096 bytes or 1024DW is used the efficiency becomes 1024/1029

which is 99.5%. Figure 18 contains a graph of throughput as a function of TLP size given by the equation (P/P+5) where P is the payload in data words.

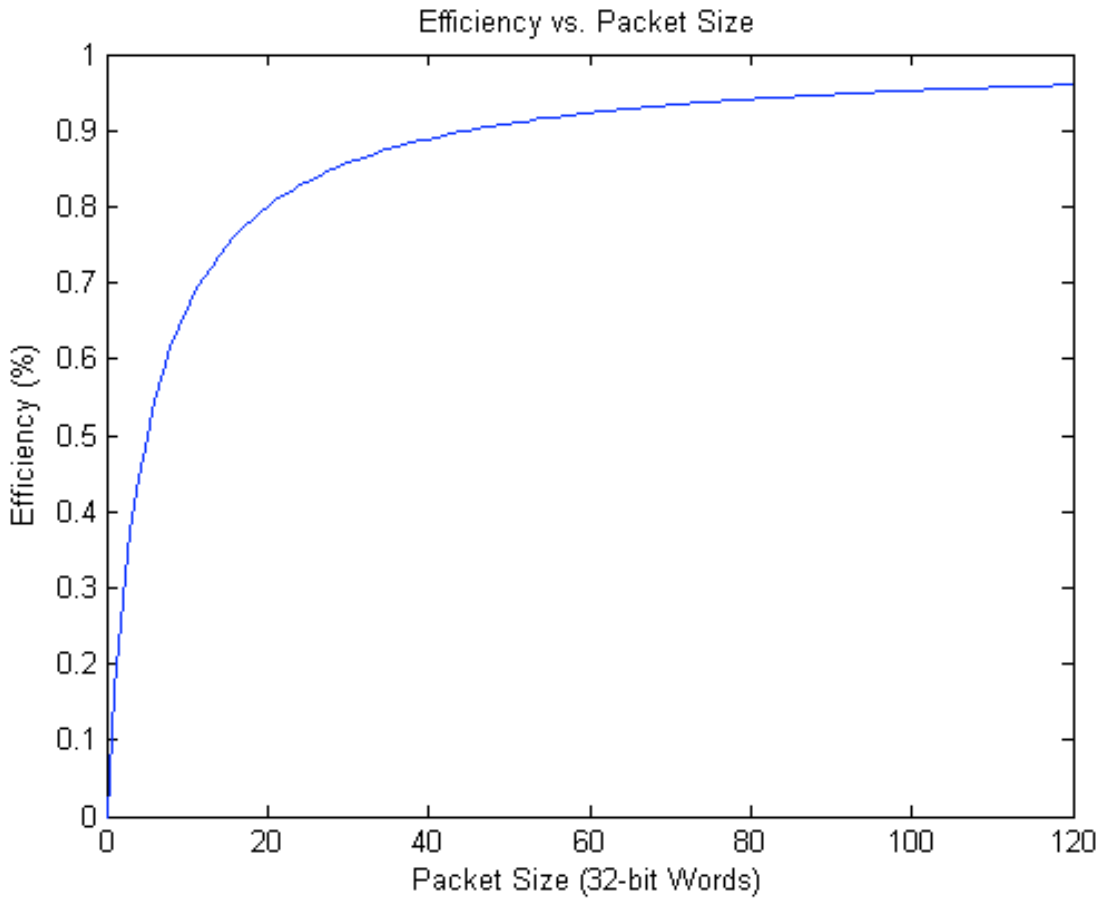

Figure 18: Graph of efficiency vs payload size

5.5 Demo Filter Implementation

5.5.1 FIR Filters

We chose to implement a finite impulse response (FIR) filter. This is a filter where delay line taps are multiplied by foxed constants and then fed to the output. We chose a FIR finter instead of an IIR (infinite impulse response), which employs loopback, because a FIR is more well behaved when its coefficients are quantized.

5.5.2 FIR Filter Implementation

For the demo we decided to implement a 4 and a 42 tap FIR filter on the FPGA using dsp48e slices. The dsp48e slice is a multiplier paired with a multi-function ALU and routing. They are stacked in columns with inter-column interconnections allowing them to run in chains at 500mhz. For a graphical representation, see figure 19.

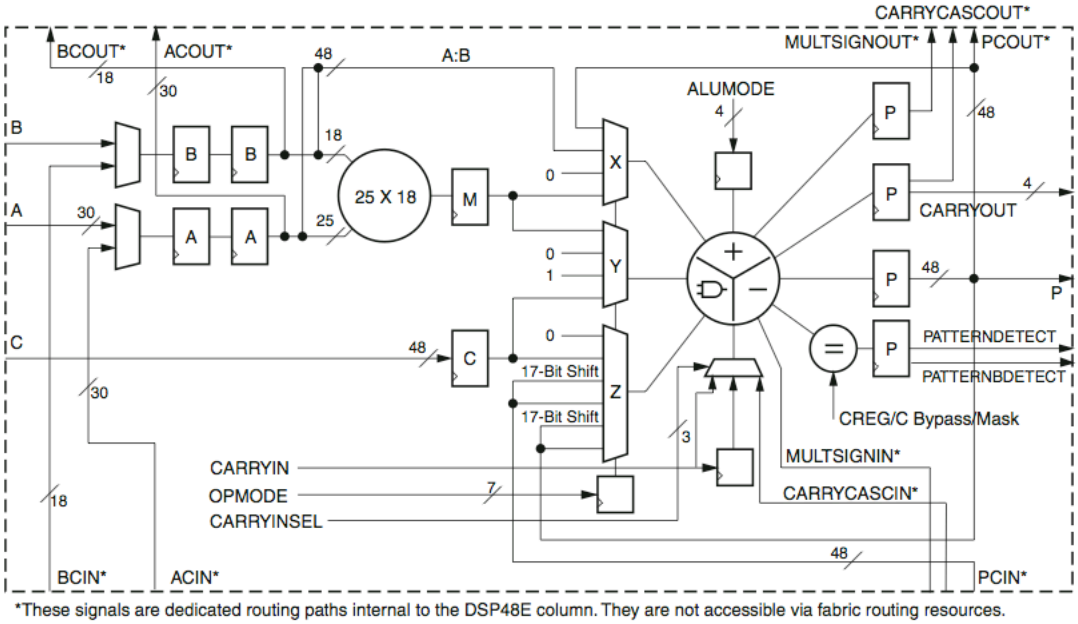

Figure 19: A diagram showing the architecture of a DSP48e slice [Xilinx ug193 p.16].

Initially, when developing the filter we opted for the transposed FIR design for efficiency reasons because the transposed design shifted the delay blocks into the adder chain, pipelining it and allowing for a higher operating frequency. However we learned that the alternate solution of tapping a delay line (figure 20) was also feasible, because a delay line could have been implemented using the ACOUT (A input Carry OUT) cascade lines. Our solution utilised the PCIN (Product Carry IN) and PCOUT (Product Carry OUT) inter-module connections with the ALU configured to sum the PCIN input with the result of the multiplication.

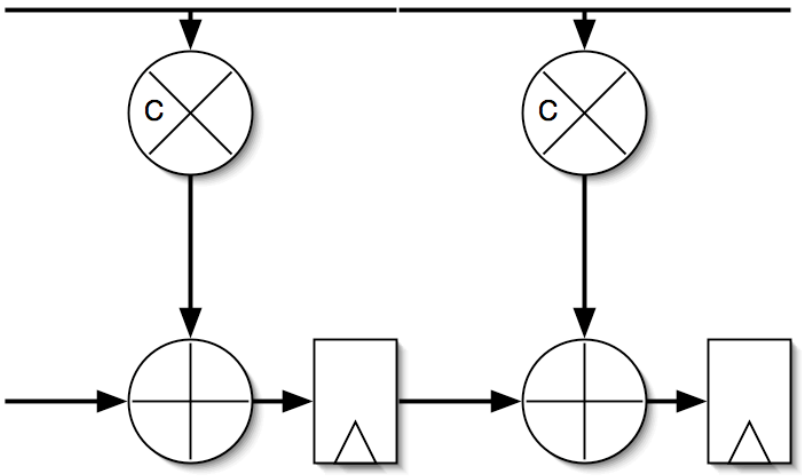

Figure 20: A data flow diagram of two cascading FIR blocks in the filter

For the initial filter we only used a small fraction of the column but for the RRC filter we used 42 out of 49 DSP48e slices.

Chapter Summary

Most of our time was spend on trying to get the FPGA board to communicate with the computer and to get an example working. We succeeded in writing a driver, assembling a bitmap, and creating a demo to show out project processing data that originated in the host. As part of future work, we need to optimize the components to increase performance and implementing dynamic reconfiguration.

6. FPGA/GNU Radio Integration

We attempted two levels of FPGA integration into GNU Radio. The first was loopback, with the FPGA simply taking data from one point in the pipeline and being read from in another point in the pipeline to retrieve the same data. The second consisted of two filter applications being executed on the FPGA: a lowpass filter, and a raised cosine filter.

6.1 Loopback

Some alterations to the original design had to be made in order to accommodate for how the driver was designed. The input and output device nodes expected in the code had to be specified as being the same node, and packet construction had to be changed from taking a string from user input to reading from a file with a previously saved string within it. Text transmission with FPGA loopback was considered theoretically functional, with slight defects. All of the characters in the desired string, “hello world” (stored as text in a file), were received in the end. However, the characters appeared out of proper order, starting in the middle and continuing through the entire string from there. Resulting outputs included “orldhello w”, “ worldhello” and “o worldhell”. The GNU Radio code for relaying text back and forth between computers is somewhat similar to previous data transfer code, and has already been somewhat thoroughly tested as working on its own. Furthermore, debugging by printing the string to terminal from the transmitting computer has shown that the string that is originally transmitted matches the received string. Thus, the problem lies in reading from or writing to the FPGA, either in the driver or in the GNU Radio handling of the device node. This was a problem resulting from the

last word entered being the first word read, even when it was part of a previous packet. Word carry-over can be disabled, and has been, resulting in functional text transfer.

6.2 Lowpass and Raised Cosine Filters

Implementing a Viterbi decoding and convolutional encoding demo as well as a filtering demo, as initially planned, proved to be too time-consuming. Dividing our efforts between multiple demonstrations during the final two weeks available to us would result in a lack of time allotted to debugging and benchmark testing. Furthermore, the concept of filter is more familiar overall, and the results are more easily quantifiable when FPGA integration is compared to GNU Radio-only implementation.

6.2.1 System Design

The filter structure itself consists of an arrangement of DSP48e slices. It is designed to handle 16-bit integers. The GNU Radio environment passes the filter floats, so proper float-to-int conversion is required. Rather than conventional floating point numbers, we implemented binary scaling. This involves some degree of rounding, scaling both the floats and filter coefficients by 65535 (16 bits). This, at first, resulted in an alteration in filter behaviour, shown in figure 21.

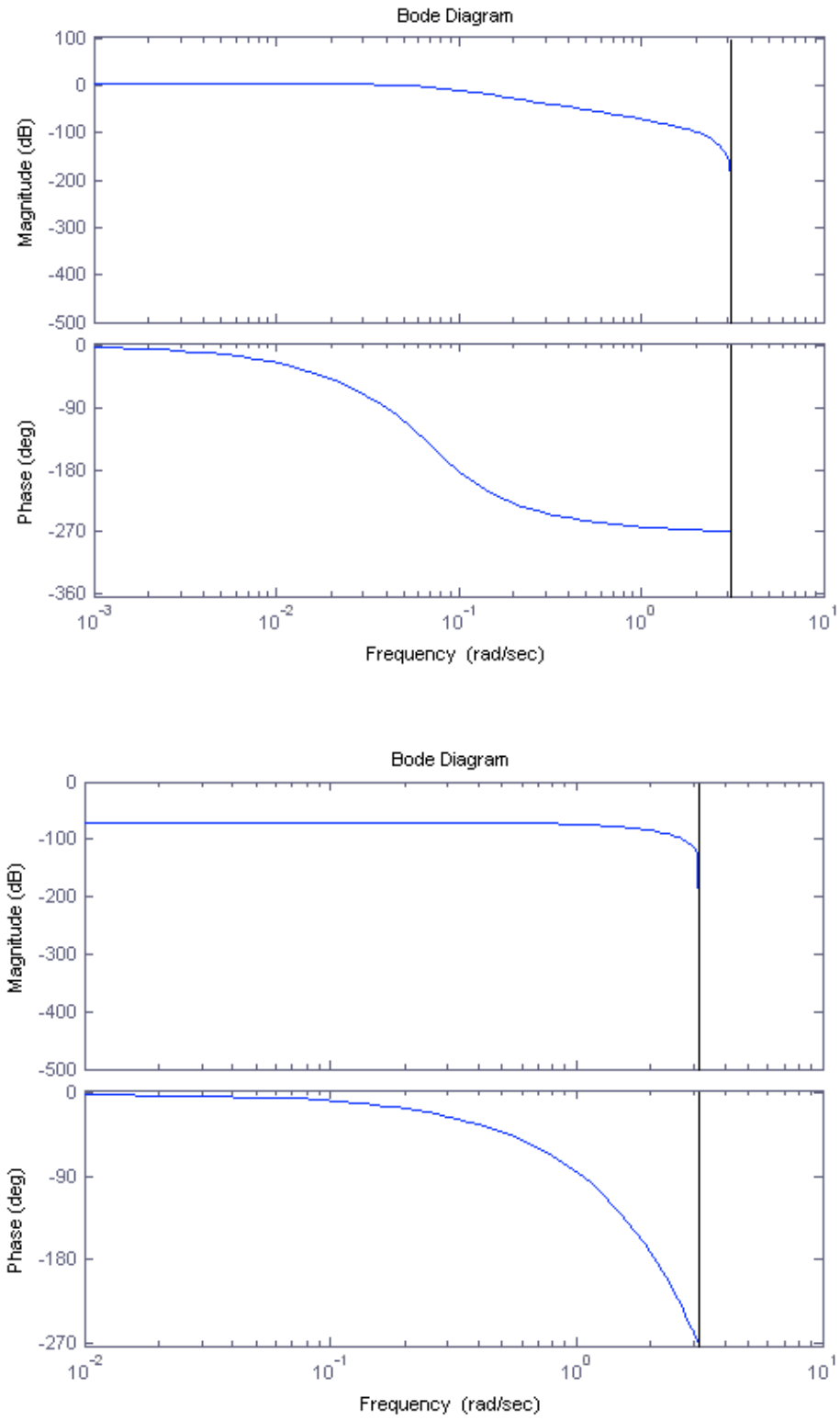

Figure 21: 4th Order Low-Pass Filter, with MATLAB-calculated coefficients (left, B =1.0e003 *[0.0379 0.1138 0.1138 0.0379], A = [1.0000 -2.8626 2.7344 -0.8716]) and scaled/rounded coefficients (right, B = [2 7 7 2], A = [1 0 0 0])

The phase differs, as shown, but given that none of our signal processing is phase-dependent at the moment, this is tolerable. There is an issue with this approximation, namely that the attenuation remains at a near-constant -60dB until cutting off, whereas the MATLAB designed filter begins at 0dB and gradually decreases. The consistent attenuation may be compensated for by scaling the signal that passes through the filter, but performance still needs to be evaluated.

6.2.2 Comparison to GNU Radio only

Unlike the FPGA-integrated FM transmission with output filtering, the GNU Radio-only implementation has different processing blocks than file sinks and sources for the handling of the filter. The first generates the proper filter coefficients (optfir.lowpass()) and the next implements the filter itself (gr.fir_filter_fff()). The FIR filter is integrated into the receiving pipeline, and the results of the FM demodulation are filtered. With a transmitter configured to transmit three sine waves on the same FM frequency, the low-pass filter (given the proper coefficients) will isolate just one of them.

The GNU-Radio exclusive code is sound in theory, and transmitter and receiver each run without errors, but confirmation of complete functionality reached a brief obstacle that held things back approximately two days. The receiving program for FM filtering requires a sound card to have observable results. The hardware is built into our workbench computer, but requires recompiling the kernel to be supported. Reception is required to be on that computer, since it is the computer with the FPGA installed. Theoretically we could swap the transmitting and receiving computers for GNU Radio-only implementation, but YunLing also does not readily support the GUI. This was resolved by replacing the audio sink with a file sink in the receiver program, and running the transmitter on Wachusett.

As a workaround for FPGA testing while rearranging GNU Radio FM transmission, we have created a loopback pipeline which generates three sine waves, sums them, passes them through a GNU Radio filter (later the FPGA) and stores them in a file. This file can be used as proof of concept for FPGA integration, FPGA-integrated FM transmission with filtering could be implemented in future works, once FPGA filtering is confirmed functional. Since full FPGA integration functionality was not assured until the end of the project, this would ideally be an initial demo in any future work.

When transmitting three summed sine waves of different frequencies and filtering above the lowest frequency but below the second lowest with a lowpass filter, we expected to receive exclusively the lowest frequency sine wave. This was not the result, however. When played back on speakers, it was a series of periodic clicking noises we received, not a tone. We also stored the results in a file, so we could reproduce and plot the output. The waveform in the time domain is shown in figure 22.

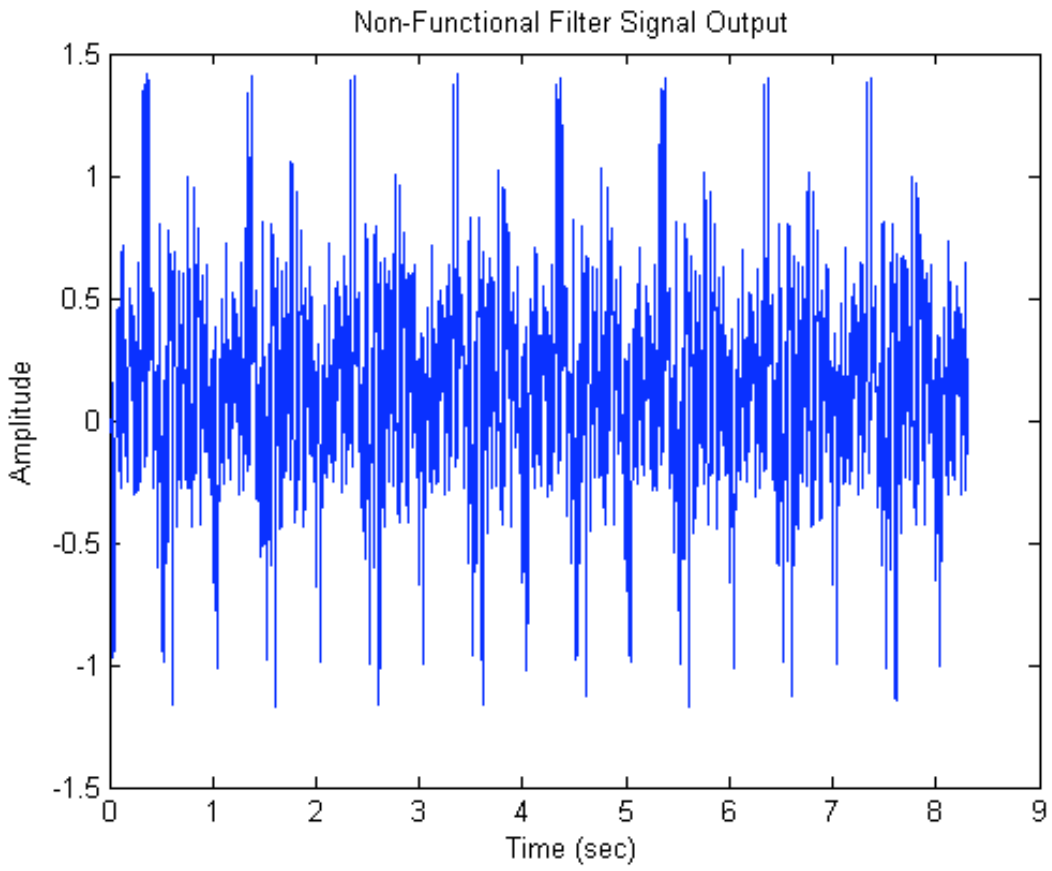

Figure 22: Output of our filter after being supplied a sum of sine waves. It appears to be noise.

This coincides with the sound we were hearing. It still looks roughly periodic (in that there are primary and secondary spikes at similar locations). To get a better idea of how the filter was behaving, we looked at the output in the frequency domain. The results can be found in figure 23.

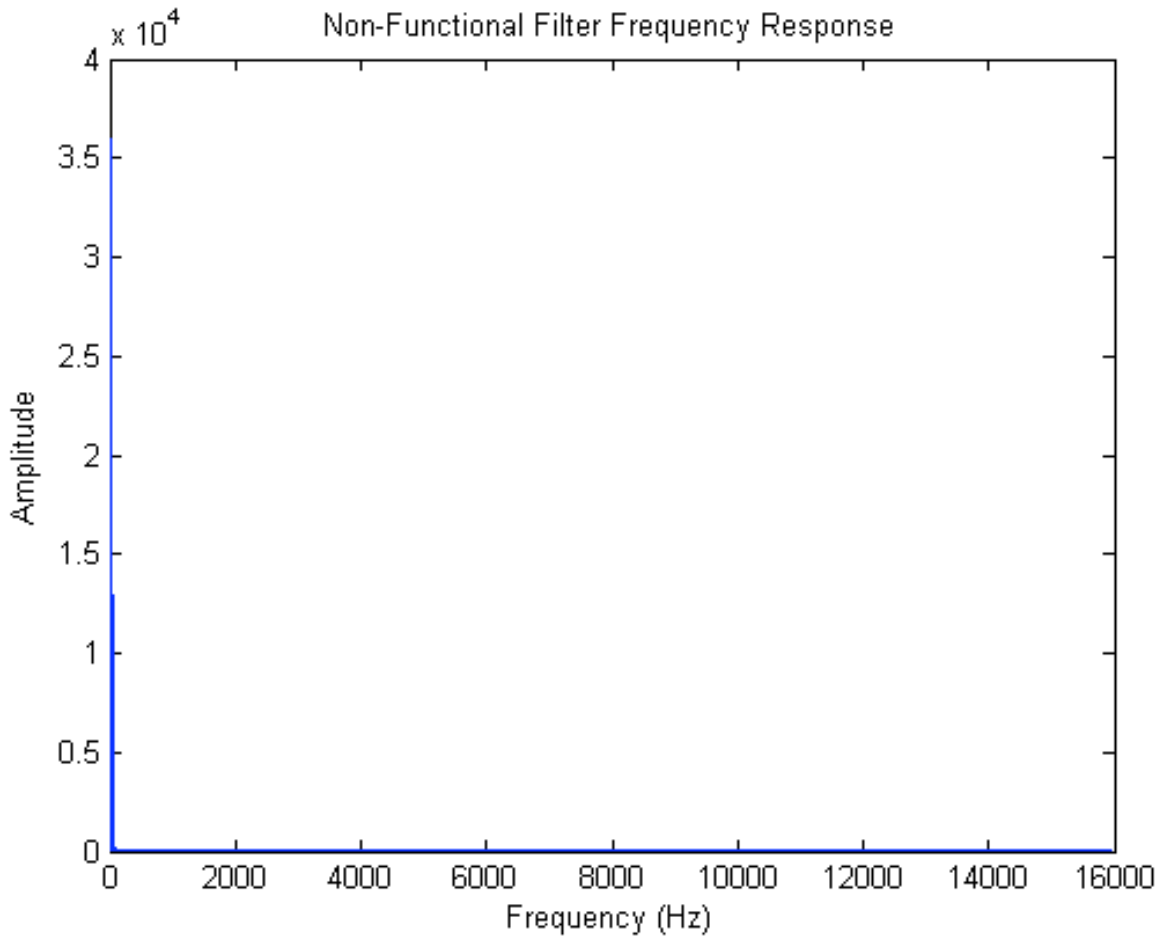

Figure 23: Frequency response of ineffective filter test. As expected from the time domain output, all but low-frequency noise is attenuated.

It can be presumed from here, and was confirmed by getting a closer look, that the filter cuts off at a very low frequency (as it turned out, 100 Hz). This was less than the 350 Hz cutoff frequency by a fair margin.

To get a comparison, and to confirm that the filter was not simply skewing everything, we left the post-modulation filter out and ran the summed signals over FM. The result was, again, not as expected (figure 24).

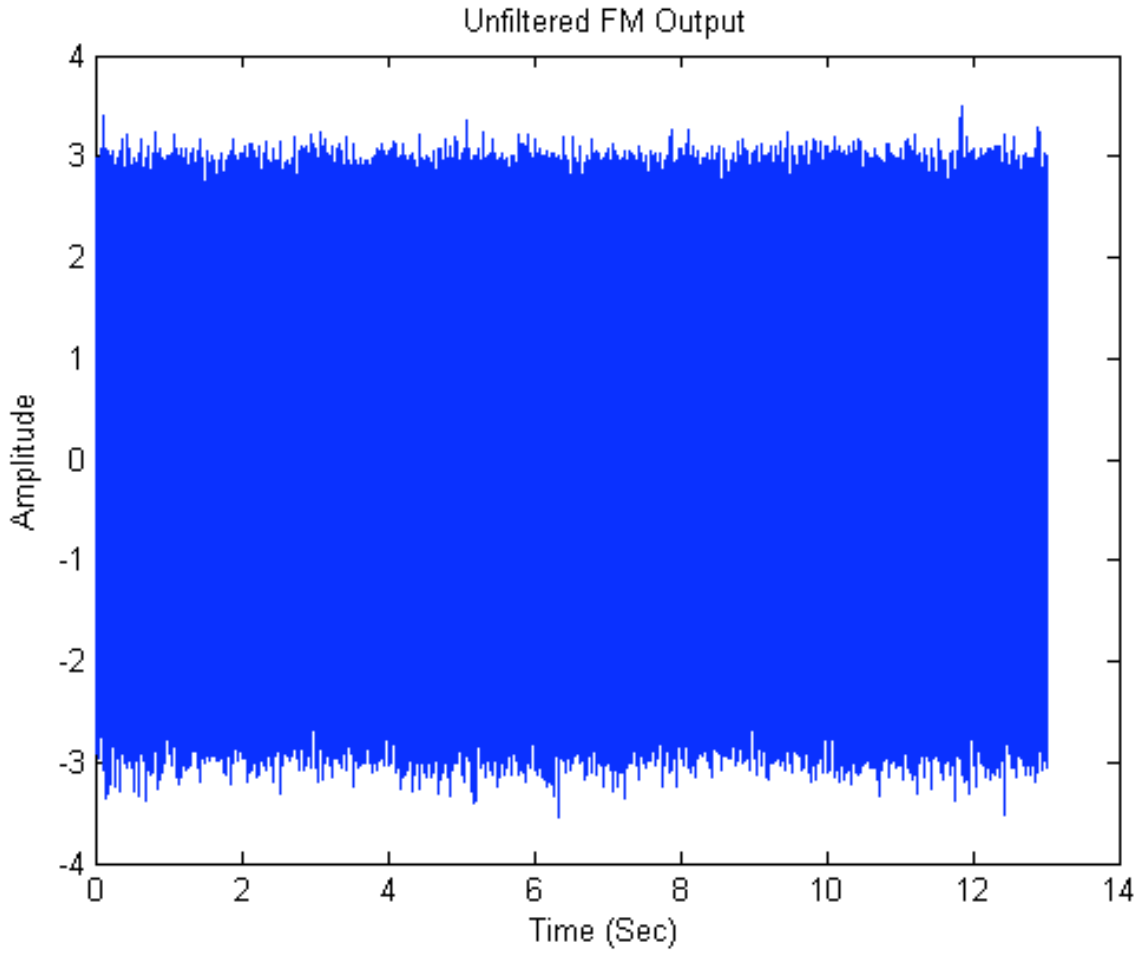

Figure 24: The unfiltered output of the sum of three sine waves varying in frequency. It appears to be high-amplitude noise.

Rather than the sum of three sine waves of varying frequency, we get noise. We do not know what would cause this result. Excessive noise, perhaps, but it was transmitted over copper wire and it should not be this severe.

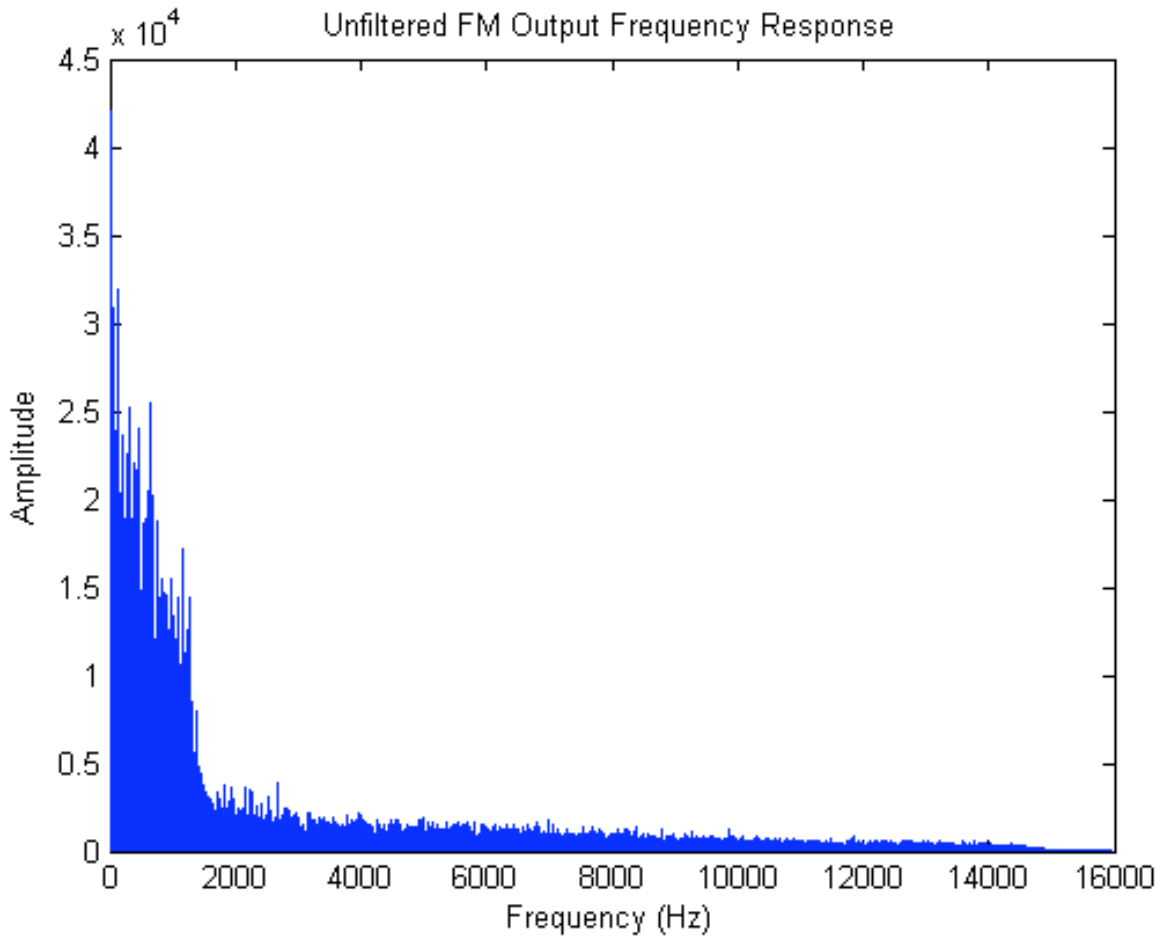

Figure 25: Frequency response of unfiltered signal. Again, appears to be low-frequency noise. The noise still appears confined to the frequencies of the sine waves used as the input signal, but there are no distinct peaks present.

Frequency response (figure 25) portrays the same. There should be three distinct spikes of equivalent amplitude, but there is instead low-frequency noise where the signals should be. As this presents the possibility that the signal frequencies chosen were too close together, a final attempt was made, using a wider range of frequencies (spaced apart at approximately 1000 Hz intervals.

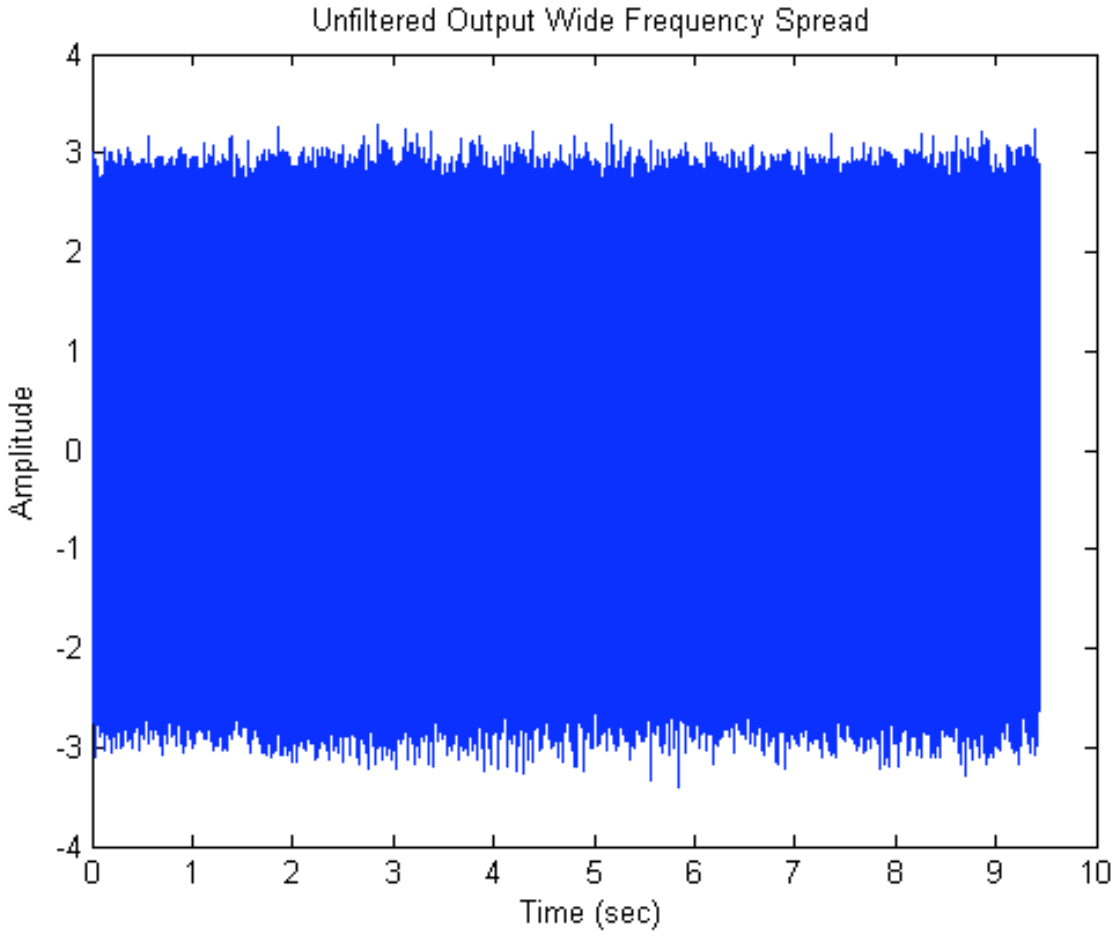

Figure 26: Another summed signal, consisting of three sine waves with a wide range of frequencies rather than similar frequencies. In the time-domain, there is no noticeable difference.

The time-domain output (figure 26) is effectively identical to frequencies that are close together.

To distinguish the difference, we need to look at the frequency domain.

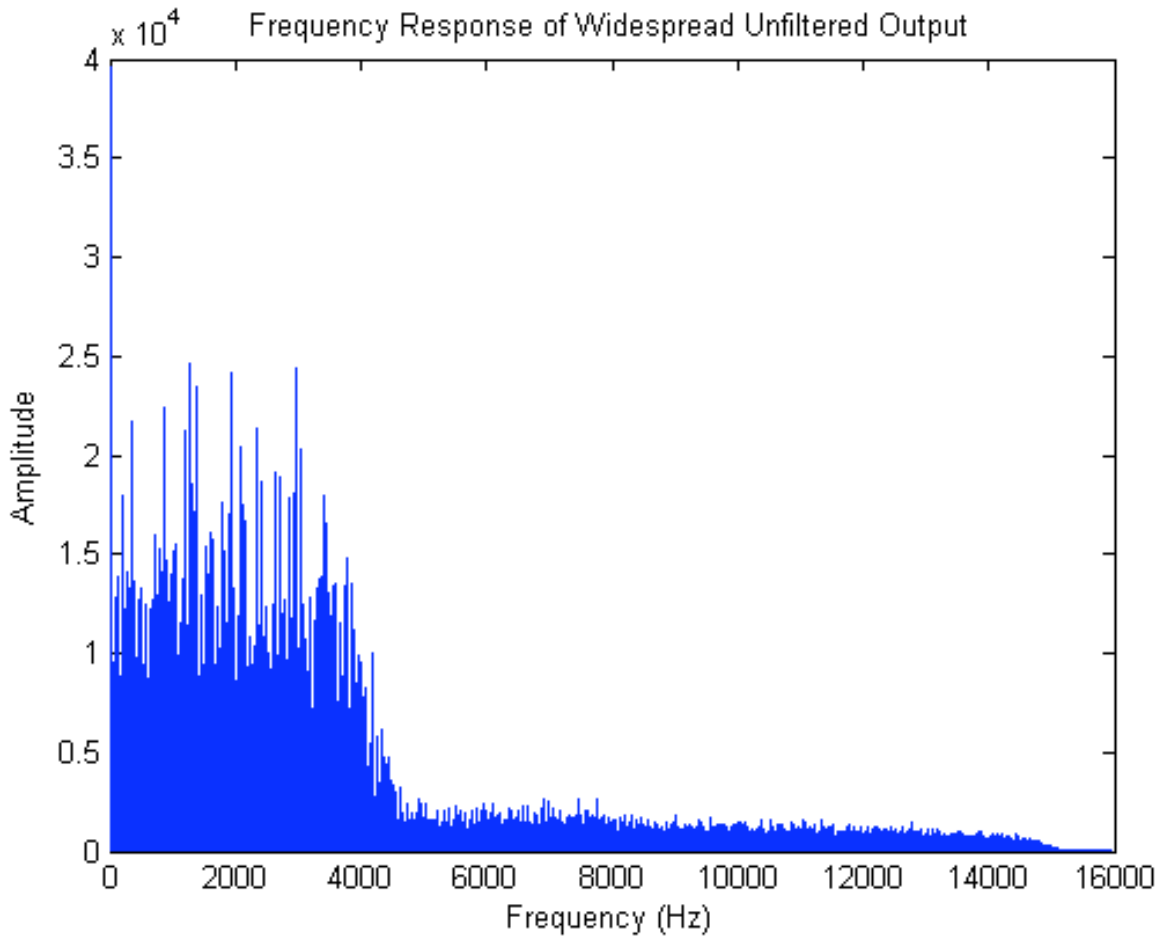

Figure 27: Frequency response of widespread frequency signal. Still seems like noise, but it is spread over more frequencies.

Figure 27 shows that there is still noise spanning the range of the sine wave frequencies, and there are no distinct peaks at the three sine wave frequencies.

Later we realized that the filter was functioning properly, but none of the signals were within the range of the filter's passband. Knowing this, we moved on to alternate testing signals, such as impulse and white noise (figures 28 and 29 respectively). Also, for the sake of reliability, we implemented the filter entirely on one computer rather than over USRP, using a single GNU

Radio pipeline to send input values, implement the filter, and store output values. These had noticeable results within the passband of the filter.

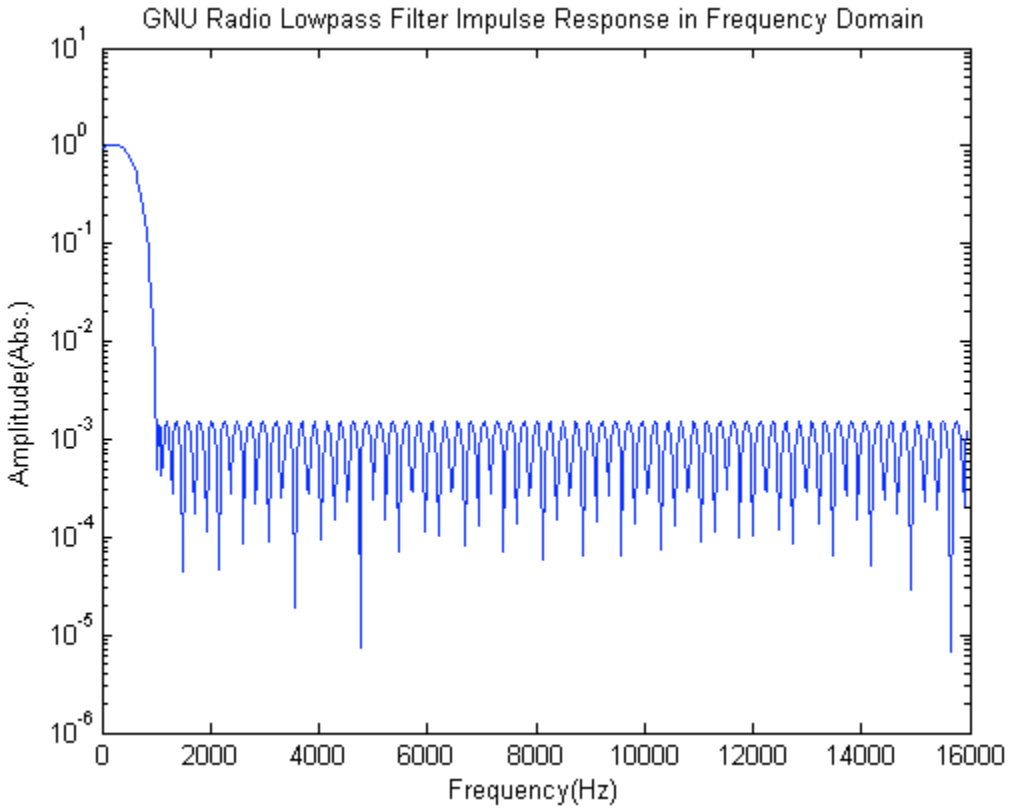

Figure 28: The impulse response for the lowpass filter we implemented entirely within GNU Radio. GNU Radio designs filter coefficients for optimum performance with the optfir processing block, and in this case specified 133 coefficients (a 132nd order filter). It has the following characteristics: Sampling frequency 32000 Hz, passband cutoff 340 Hz, stopband cutoff 1000 Hz. This was the filter expected to isolate one of the sine waves in the sum of signals from the prior attempt, but it failed to do so. This is likely because the frequency response of the signals lies unexpectedly beyond the non-attenuated range of the filter.

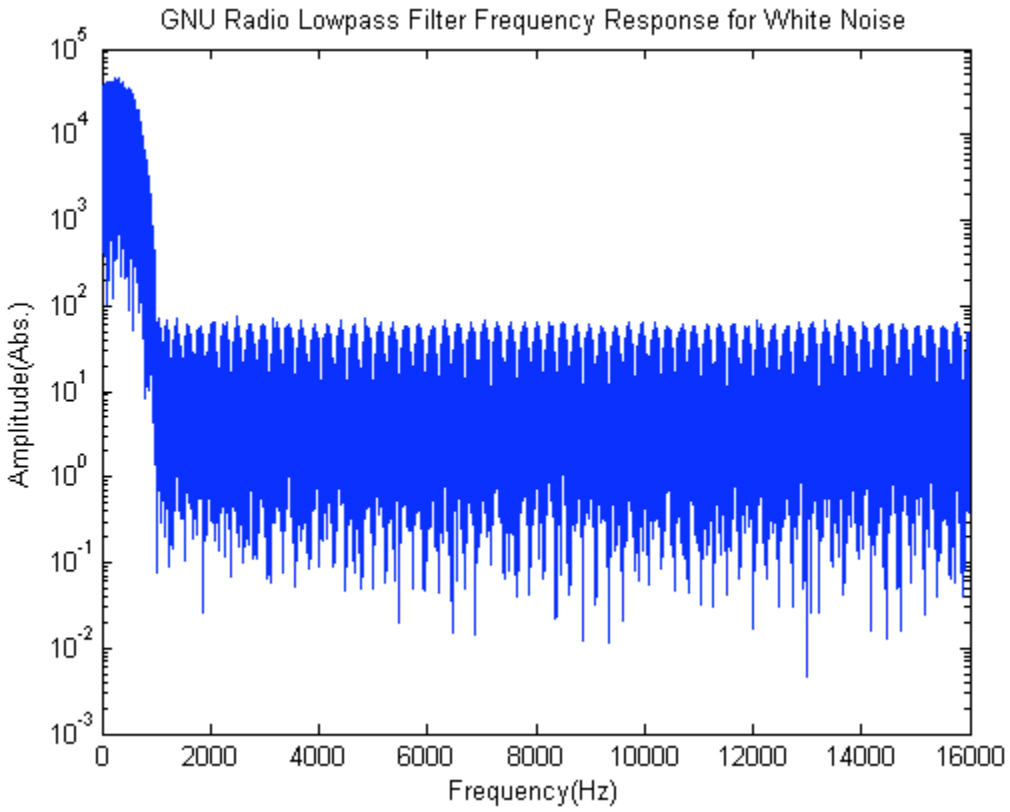

Figure 29: The frequency response of the GNU Radio lowpass filter, given whit e noise. The attenuation pattern matches the impulse response, but the amplitude is higher in magnitude. This is because the output here is the product of the input (white noise), and the impulse response of the filter. It is attenuated relative to the impulse response, but its amplitude is proportional to the amplitude of the white noise supplied to the filter. Overall, it functions as expected.

6.2.3 FPGA Filter Testing

Prior to testing the FPGA filter, we decided to make alterations to the filter we were using, since we were having trouble initially determining coefficients. We opted for a higher frequency cutoff for the filter we chose, since its effect would be more readily observable. The passband cutoff was moved up from 340 Hz to 8000 Hz, as will be apparent in the later plots.

Over the course of the last full week of C-term, the first attempt at an FPGA filter demo was made. During debugging of the filter hardware, a program called DD was used for supplying

input to the filter and reading its output; but for the purposes of the demo, we used GNU Radio to perform these tasks. This provided the bare minimum for a demo. The filter being implemented used only four taps. Furthermore, the FPGA could not both read and write simultaneously with the existing drivers. The demo implemented two GNU Radio pipelines: one for writing floats to the FPGA (an impulse, or white noise), and one for reading from the FPGA and storing the results in a file. To plot the results and not have raw hexadecimal output, a conversion script was required to make the floats acceptable for MATLAB's `textread` function. From there, standard MATLAB conventions can be used for plotting filter output. Furthermore, to prove that the output was approximately correct, we implemented a filter with the same coefficients in MATLAB outright and plotted the results. The frequency response for a delta function is shown in figure 30, and figure 31 shows the frequency response to white noise.

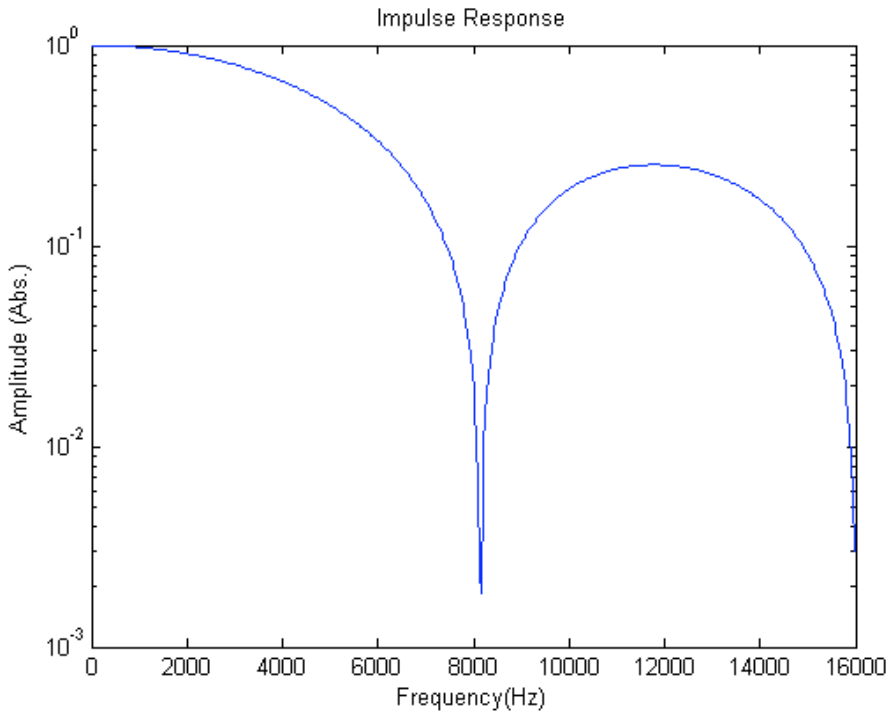

Figure 30: Frequency response of the 4th-order lowpass filter to a delta function. The following plots reflect the form of this impulse response.

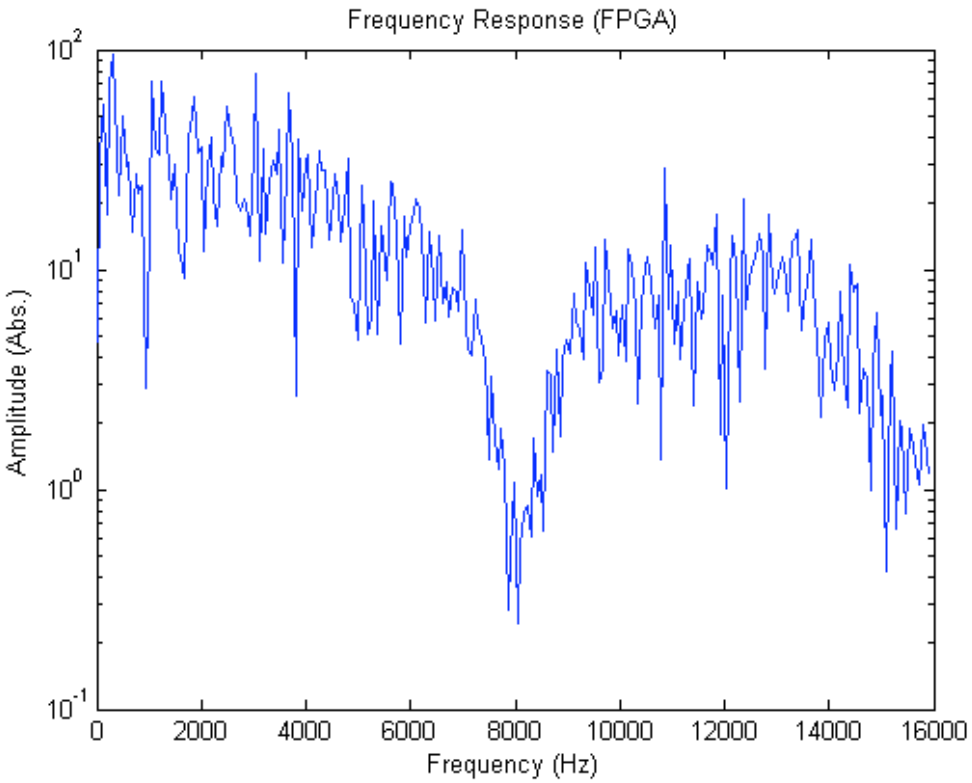

Figure 31: Frequency response from our FPGA filter. It appears less dense than the MATLAB-implemented filter due to the acquisition of fewer data points, and the amplitude is different because it depends on the amplitude of white noise passed through the filter at the time. It is the form of attenuation change that is important.

For the sake of comparison to more dependable values, we implemented a filter with the same coefficients in MATLAB. We chose MATLAB because it provides greater control over filter qualities than GNU Radio does. GNU Radio has a processing block called `optfir`, which calculates the optimal filter it can handle. There is no parameter for the order of the filter, which is critical for a reliable comparison. In higher-order filters this would not make much of a difference, but in low-order filters the form the filter takes varies very significantly between orders. Given the coefficients we produced, MATLAB provided a plot with a similar form to our filter. The MATLAB plot is shown in figure 32.

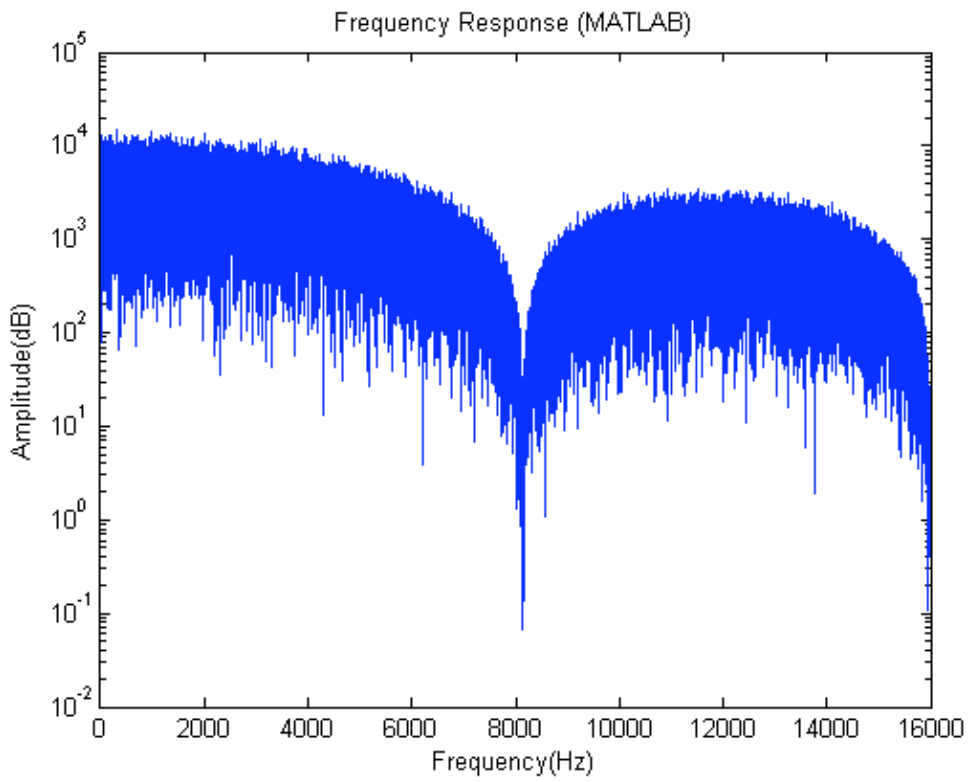

Figure 32: Frequency response of a filter with similar coefficients in MATLAB.

Some technical difficulties while getting this far included the lack of decimal point in a parameter for the GNU radio multiply block, which caused skewed the output to consist entirely of the word FFFF repeating. This much was quickly remedied, but there was another technical issue which left its mark on the entire demo – the FPGA was incapable of being written to and read from simultaneously with the existing driver. This was a timing issue which was worked around temporarily by manually swapping between reading and writing (kept in separate applications). This works to test basic functionality of the filter, but to actually be practical, the timing issue needed to be remedied. Alex investigated the driver to resolve the issue in the coming days. Refer back to chapter five for a full description of driver development.

6.2.4 GNU Radio and FPGA Bidirectional Communication

Below is a short GNU Radio pipeline implementing the new driver. Its purpose is to write an impulse response to a character device node representing an FPGA (which contains a 4-coefficient lowpass filter), read the FPGA's output, and store said output in a file.

When run, the program appears to make the proper function calls to the FPGA, and relays the proper data. This is made evident by the output of the debug comments from within the driver, shown in /var/log/messages. Table 2 shows the non-zero results of sending a delta function through the fourth-order filter. The filter coefficients seem correct, so the pipeline appeared to be operational.

Filter coefficients * 65535^2 (Hex)	Filter Coefficients (Dec.)
0x4678F800	0.275291738
0x46838000	0.275452437
0x46838000	0.275452437
0x4678F800	0.275291738

Table 2: The four terms resulting from the impulse response of the filter (retrieved from the debugger). They must be divided by 65535 to accommodate for the output being offset, and then again to get the original coefficients. The second division is needed because the coefficients in the filter are scaled up since all input signals are multiplied by 65535 for float-to-fixed conversion. The divisions take place in a GNU Radio multiply_const_ff block, but the divided numbers are not shown in the debugger.

However, the pipeline at this stage just creates a destination file and does not write to it, leaving a file of 0 bytes. The output of the debugger indicates that the proper values are being requested and retrieved. The data is just not getting from the FPGA to its final destination properly. File IO functioned properly with an earlier version of the driver (which did not support simultaneous read/write). Still, this driver *can* read and write simultaneously. At first the fork() function in Python's OS module was thought the key to this, but it was later recommended by Eric Blossom that we seek alternate means. Another function in the OS module, popen(), would be a suitable

replacement that functions properly, but this could not be verified functional until the file IO issue is resolved. Given the time constraints of the project, this was an issue that could not be remedied. Possible causes of this are explained further in the Future Work chapter.

Another, more complicated filter design was implemented on the FPGA to show flexibility in the filter implementation architecture. Adding more filter coefficients is a matter of adding filter taps (each being a line of code in VHDL). As was the case with the 4^{th} order filter in the section prior, the filter coefficients for more complex filters can be calculated elsewhere (such as MATLAB) and added in the VHDL filter, mimicking the behaviour of the designed filter. For our more complicated example, we implement a 40^{th} order raised-cosine filter, with sampling frequency at 32000 Hz and passband cutoff at 8000 Hz.

The raised-cosine filter proved to be more drastically affected by quantization than the lower order filter appeared to be, due to conversion from float to fixed-point used integer multiplications and divisions rather than floating-point operations, dropping off more decimal places than would be lost otherwise. The error becomes apparent with the impulse response of the filter, which dissipates in a similar manner to what is expected, but the rebounds afterward that should be smooth curves are jagged. The imprecision due to quantization could be remedied by our float-to-fixed conversion using more decimal places. But for that would require hardware architecture restructuring, so for the sake of this demonstration, we will accept the margin of error. The impulse response of our FPGA filter is shown in the figure 33.

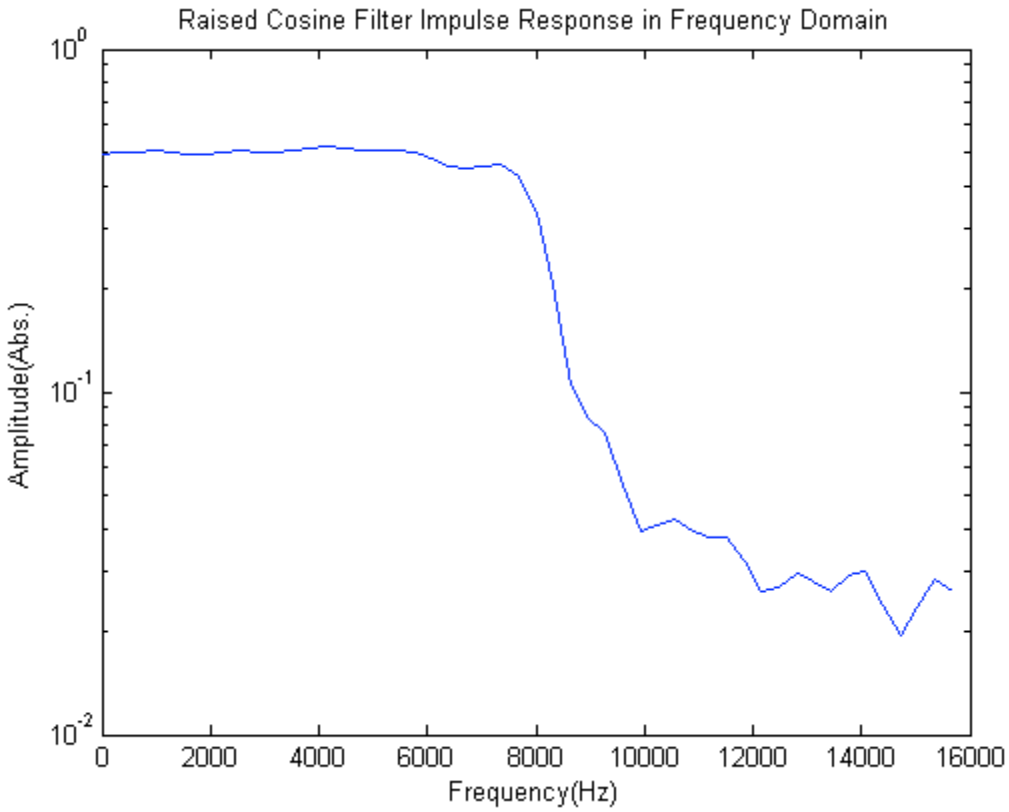

Figure 33: Frequency response of FPGA filter to a delta function. Notice jagged response at higher frequencies, most notably at 10000 Hz and higher. This is stopband ripple, but the curves do not appear as smooth as they should be. The jagged nature of the output is attributed to quantization error.

The frequency response of white noise follows the impulse response provided above relatively well, as is seen in figure 34. For the sake of comparison, we will also show the output of a filter of similar coefficients in MATLAB if supplied the same sample of white noise (figure 35).

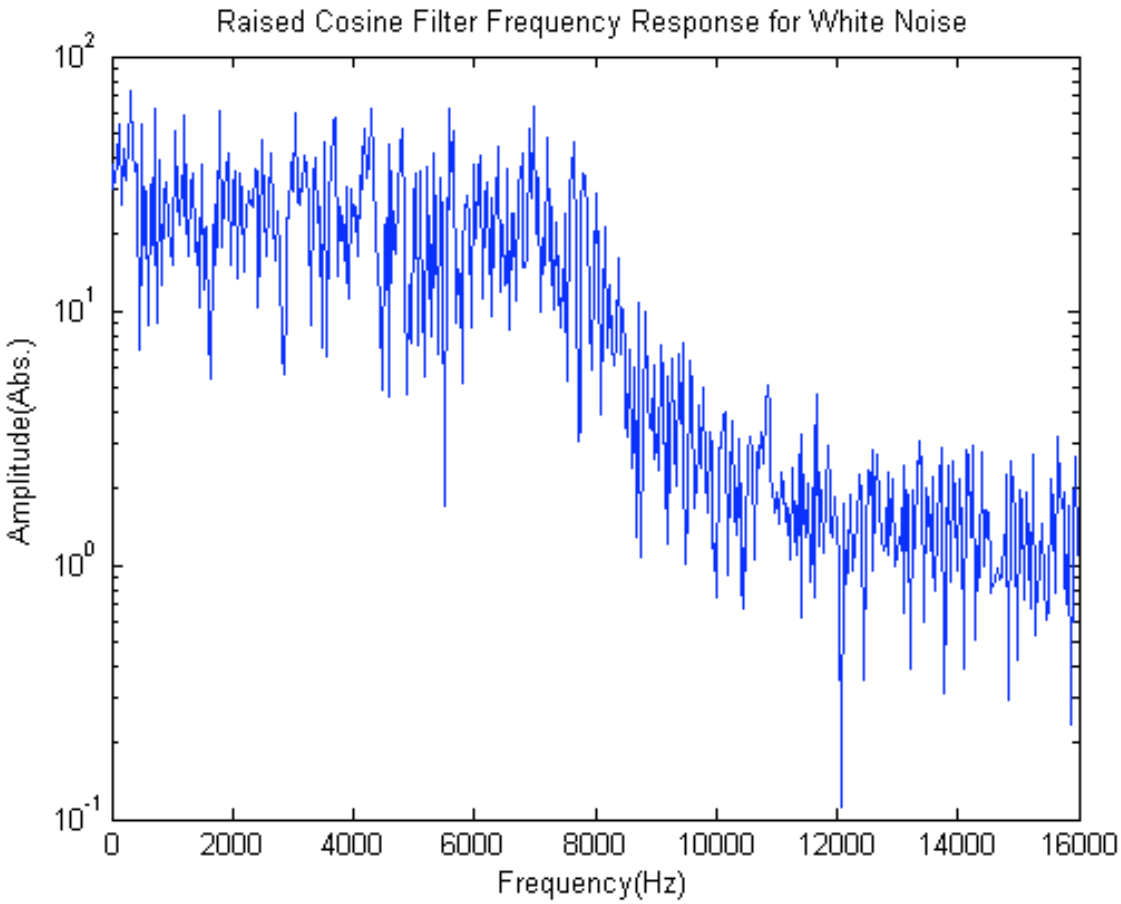

Figure 34: Frequency response of white noise after being passed through our raised-cosine filter. It is apparent that this waveform follows the same pattern as the impulse response, dropping off after 8000 Hz.

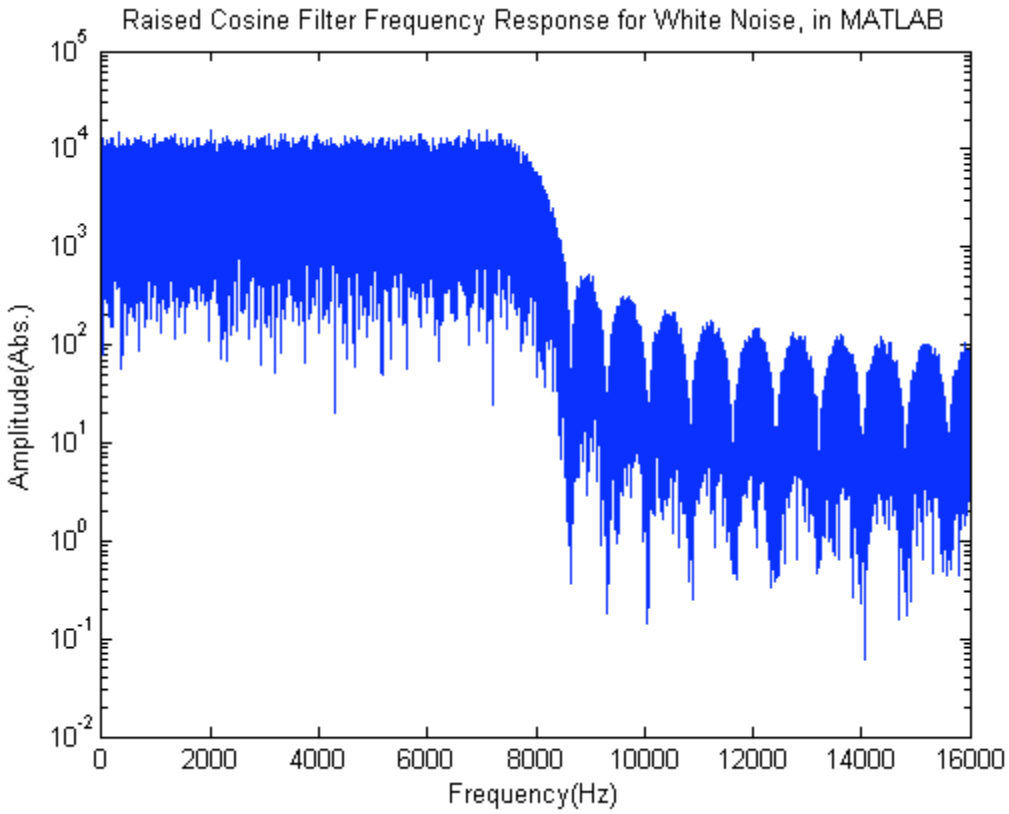

Figure 35: Frequency response as determined by MATLAB. As was the case before, there seems a higher density of data points. Also, the difference between pre- and post-quantization is readily noticeable after seeing both frequency response plots.

With this filter, we executed some performance testing. By timing the execution of passing a known number of zeroes through the filter, then writing from the filter to /dev/null, we were able calculate the bitrate for the filter given the current hardware driver. This manner of testing indicated that the current driver is a severe bottleneck, producing only 63.3 kB/s when reading and writing simultaneously, and 2 MB/s when not streaming constantly. This is sub-par for typical filter hardware, and could be remedied with driver modifications. It explains the appearance of fewer samples in our FPGA plot than in our MATLAB filter plot. The driver we are currently using is the bare minimum for functionality, and should be modified in future work. However, the basic concept is sound and can serve as a basis for other objectives.

Chapter Summary

In summary, current status of GNU Radio integration is that it can write to FPGA and read from FPGA. However, the file sink for the final destination data does not function properly. It generates a destination file, but does not write to it. It is likely that this problem could be remedied with changes either to the driver or to the GNU Radio file sink processing block. For continuing work, we recommend that this bug be addressed first. Additional suggestions for continuing work are provided in the next chapter.

7. Future Work

There are a couple of key points that were once objectives of the project, which were dropped due to time constraints. The first issue that should be addressed is the output aspect of file I/O in the GNU Radio/FPGA filter pipeline. Since the implementation of bidirectional reading and writing with the filter architecture, the file sink has not functioned as expected. The destination file specified is created, but no data is written to it. As described in Section 6.2.4, the hardware has been verified functional according to the debugger. There is likely a small bug within the driver's write function that does not affect operation, but does affect its interaction with GNU Radio's file_sink block. So the I/O problem could be resolved with either driver modification or a workaround in GNU Radio to avoid using the file_sink block. However, we were not able to deal with this problem due to lack of time. This should be the first matter to be dealt with should anyone choose to continue the work of this project.

Aside from the I/O problem, streamlining the driver is very strongly recommended. Rewriting the driver to accept data in bulk rather than individual words per packet would drastically increase efficiency. The current driver, while functional, has a ratio of applicable data to header data in its packets which is not viable in a practical application, and it shows in the noticeably less dense collection of data points in our filter output plots. The hardware driver should be the focus of any further work at first.

On the side of FPGA hardware architecture, the float to fixed-point conversion could be improved, to increase the decimal point accuracy of quantized filter coefficients and more accurately generate filters. It is still accurate enough as-is to get the general proof of concept, but

for practical applications, using floating point numbers in the multiplications and divisions for float-to-fixed conversion rather than integers is recommended. .

Another possibility that could stem from the outcome of our project would be to pursue implementation of dynamic reconfiguration within the FPGA. This was one of the initial desires of the project. But dynamic reconfiguration got put on hold, once again due to time constraints and a need to get the hardware architecture of the FPGA functional to an acceptable level.

8. Conclusion

Overall, some of our initial goals were accomplished over the course of this three-term project. While we did not meet our initial goal of dynamic reconfiguration of our FPGA within GNU Radio, we did provide mostly functional integration of an FPGA into GNU Radio pipelines with both loopback and filtering applications. We have set up a foundation for future research into dynamic reconfiguration for SDR. The process will be made far easier by already having an FPGA integrated into an SDR pipeline and performing signal processing tasks in place of the host PC.

This project has been an exercise in the use of GNU Radio, signal processing, and hardware driver design. It would be advisable to have some background knowledge in these areas, prior to continuing the work of this project, or to be prepared to research these subjects early on. A fair portion of project work this term was trial and error while learning how to proceed in these fields. For future work, we recommend addressing the file I/O bug, streamlining the device driver, improving decimal point precision within the FPGA-implemented filter, and implementing dynamic reconfiguration on the FPGA.

9. References

[1] R. I. Lackey and D. W. Upmat. “Speakeasy: The Military Software Radio.” *IEEE Communications Magazine*, Vol. 33, No. 5, Pgs. 56-61, 1995.

[2] P. A. Eyermann. “Joint Tactical Radio Systems-A Solution to Avionics Modernization.” *Proceedings of 18th Digital Avionics Systems Conference* (St. Louis, MO, USA), Vol. 2, Pgs. 9.A.5-1 - 9.A.5-8, 1999.

[3] W. Lehr, F. Merino, and S.E. Gillett. *Software Radio: Implications for Wireless Services, Industry Structure, and Public Policy*. Massachusetts Institute of Technology Program on Internet and Telecoms Convergence, 2002.

[4] Amara Amara, Fre´de´ric Amiel, Thomas Ea. “FPGA vs. ASIC for low power applications.” *Elsevier Microelectronics Journal,* Vol. 37, Pg. 675, 2006.

[5] J. H. Reed. *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[6] V. Bose, M. Ismert, M. Welborn, and J. Guttag. “Virtual Radios.” *IEEE Journal on Selected Areas in Communications*, Special Issue on Software Radios, Vol. 17, No. 4, Pgs. 591-602, 1999.

[7] J. Chapin, V. Bose, “The Vanu Software Radio System”, *Proceedings of the Software Defined Radio Technical Conference* (San Diego, CA, USA), November 2002.

[8] Ettus Research LLC, “The Universal Software Radio Peripheral”, URL: http://www.ettus.com/index.html

[9] Ettus Research LLC, “The Universal Software Radio Peripheral”, URL: http://www.ettus.com/downloads/er_ds_usrp_v5b.pdf

[10] R. Hosking, “Designing Software Radio Systems with FPGAs,” *Embedded Technology*, pp. 13-15, Sep. 2008.

[11] J. P. Delahaye, C. Moy, P. Leray, J. Palicot, "Managing Dynamic Partial Recon_guration on Heterogeneous SDR Platforms", SDR Forum, November, 2005.

[12] J.-P. Delahaye, J. Palicot, P. Leray, "A hierarchical modeling approach in software defined radio system design," IEEE Workshop on Signal Processing Systems Design and Implementation, vol., no., pp. 42-47, 2-4 Nov. 2005.

[13] F. Berthelot, F. Nouvel, "Partial and Dynamic Reconfiguration of FPGAs: a top down design methodology for an automatic implementation," ISVLSI , pp. 436-437, 2006.

[14] J. P. Delahaye, J. Palicot, C. Moy, P. Leray, "Partial Recon_guration of FPGAs for Dynamical Reconfiguration of a Software Radio Platform," 16th IST Mobile and Wireless Communications Summit, vol., no., pp.1-5, 1-5 July 2007.

[15] P. Sedcole, B. Blodget, T. Becker, J. Anderson, P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *IEEE Proceedings - Computers and Digital Techniques*, vol.153, no.3, pp. 157-164, 2 May 2006.

[16] E. Salminen, A. Kulmala, T. D. Hamalainen, "HIBI-based multiprocessor SoC on FPGA," *IEEE International Symposium on Circuits and Systems (ISCAS 2005)*, vol., no., pp. 3351-3354 Vol. 4, 23-26 May 2005.

[17] J. P. Delahaye, G. Gogniat, C. Roland, P. Bomel, “Software Radio and Dynamic Reconfiguration on a DSP/FPGA Platform”*Software Defined Radio of Frequenz*, May-June, No. 58, pp.152–159.

[18] A. Dejonghe, J. Declerck, F. Naessens, M. Glassee, A. Dusa, E. Umans, A. Ng, B. Bougard, G. Lenoir, J, Craninckx, L. Van der Perre, "Low-power SDRs through cross-layer control: concepts at work," IST Mobile and Wireless Communications Summit, vol., no., pp.1-6, 1-5 July 2007.

[19] Dawei Shen. Tutorial 7: Exploring the FM receiver. 2005. http://www.snowymtn.ca/ GNURadio/ GNURAdioDoc-7.pdf

[20] Dawei Shen. Tutorial 8: Getting Prepared for Python in GNU Radio by Reading the FM Receiver Code Line by Line – Part II. 2005. http://www.snowymtn.ca/ GNURadio/GNURAdioDoc-8.pdf

[21] DD invocation manual - http://www.gnu.org/software/coreutils/manual/html_node/dd-invocation.html

[22] Platform Studio and EDK documentation - http://www.xilinx.com/ise/embedded/edk_docs.htm

10. Appendices

FM Radio:

fm_tx.py:

```
#!/usr/bin/env python
from gnuradio import gr, eng_notation
from gnuradio import usrp
from gnuradio import audio
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import usrp_dbid
import math
import sys

from gnuradio.wxgui import stdgui, fftsink
from gnuradio import tx_debug_gui
import wx


##################################################
# instantiate one transmit chain for each call

class pipeline(gr.hier_block):
    def __init__(self, fg, filename, lo_freq, audio_rate, if_rate):

        src = gr.file_source (gr.sizeof_float, filename, True)
        fmtx = blks.nbfm_tx (fg, audio_rate, if_rate,
                             max_dev=5e3, tau=75e-6)

        # Local oscillator
        lo = gr.sig_source_c (if_rate,        # sample rate
                              gr.GR_SIN_WAVE, # waveform type
                              lo_freq,        #frequency
                              1.0,            # amplitude
                              0)              # DC Offset
        mixer = gr.multiply_cc ()

        fg.connect (src, fmtx, (mixer, 0))
        fg.connect (lo, (mixer, 1))

        gr.hier_block.__init__(self, fg, src, mixer)



class fm_tx_graph (stdgui.gui_flow_graph):
    def __init__(self, frame, panel, vbox, argv):
        stdgui.gui_flow_graph.__init__ (self, frame, panel, vbox, argv)
```

```
        parser = OptionParser (option_class=eng_option)
        parser.add_option("-F", "--filename", type="string",
default="audio.dat",
                          help="read input from FILE")
        parser.add_option("-T", "--tx-subdev-spec", type="subdev",
default=None,
                          help="select USRP Tx side A or B")
        parser.add_option("-f", "--freq", type="eng_float", default=None,
                           help="set Tx frequency to FREQ [required]",
metavar="FREQ")
        parser.add_option("","--debug", action="store_true", default=False,
                          help="Launch Tx debugger")
        (options, args) = parser.parse_args ()

        if len(args) != 0:
            parser.print_help()
            sys.exit(1)

        if options.freq is None:
            sys.stderr.write("fm_tx4: must specify frequency with -f FREQ\n")
            parser.print_help()
            sys.exit(1)

        # ---------------------------------------------------------------
        # Set up constants and parameters

        self.u = usrp.sink_c ()       # the USRP sink (consumes samples)

        self.dac_rate = self.u.dac_rate()                     # 128 MS/s
        self.usrp_interp = 400
        self.u.set_interp_rate(self.usrp_interp)
        self.usrp_rate = self.dac_rate / self.usrp_interp    # 320 kS/s
        self.sw_interp = 10
        self.audio_rate = self.usrp_rate / self.sw_interp    # 32 kS/s

        # determine the daughterboard subdevice we're using
        if options.tx_subdev_spec is None:
            options.tx_subdev_spec = usrp.pick_tx_subdevice(self.u)

        m = usrp.determine_tx_mux_value(self.u, options.tx_subdev_spec)
        #print "mux = %#04x" % (m,)
        self.u.set_mux(m)
        self.subdev = usrp.selected_subdev(self.u, options.tx_subdev_spec)
        print "Using TX d'board %s" % (self.subdev.side_and_name(),)

        self.subdev.set_gain(self.subdev.gain_range()[1])    # set max Tx
gain
        self.set_freq(options.freq)
        self.subdev.set_enable(True)                         # enable
transmitter

        sum = gr.add_cc ()

        # Instantiate channel
        t = pipeline (self, options.filename, 0, self.audio_rate,
self.usrp_rate)
        self.connect (t, (sum, 0))
```

```
        gain = gr.multiply_const_cc (4000.0)

        # connect it all
        self.connect (sum, gain)
        self.connect (gain, self.u)

        # plot an FFT to verify we are sending what we want
        if 1:
            post_mod = fftsink.fft_sink_c(self, panel, title="Post
Modulation",
                                          fft_size=512,
sample_rate=self.usrp_rate,
                                          y_per_div=20, ref_level=40)
            self.connect (sum, post_mod)
            vbox.Add (post_mod.win, 1, wx.EXPAND)


        if options.debug:
            self.debugger = tx_debug_gui.tx_debug_gui(self.subdev)
            self.debugger.Show(True)


    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.  Finally, we feed
        any residual_freq to the s/w freq translater.
        """

        r = self.u.tune(self.subdev._which, self.subdev, target_freq)
        if r:
            print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
            print "r.dxc_freq      =", eng_notation.num_to_str(r.dxc_freq)
            print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
            print "r.inverted      =", r.inverted

            # Could use residual_freq in s/w freq translator
            return True

        return False

def main ():
    app = stdgui.stdapp (fm_tx_graph, "Single-channel FM Tx")
    app.MainLoop ()

if __name__ == '__main__':
    main ()
```

fm_rx.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from gnuradio.wxgui import slider, powermate
from gnuradio.wxgui import stdgui, fftsink, form
from optparse import OptionParser
import usrp_dbid
import sys
import math
import wx

def pick_subdevice(u):
    """
    The user didn't specify a subdevice on the command line.
    Try for one of these, in order: TV_RX, BASIC_RX, whatever is on side A.

    @return a subdev_spec
    """
    return usrp.pick_subdev(u, (usrp_dbid.TV_RX,
                                usrp_dbid.TV_RX_REV_2,
                                usrp_dbid.BASIC_RX))


class wfm_rx_graph (stdgui.gui_flow_graph):
    def __init__(self,frame,panel,vbox,argv):
        stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)

      usage="%prog: [options] output_filename"
        parser=OptionParser(option_class=eng_option)
        parser.add_option("-F", "--filename", type="string", default="USRP",
                          help="read input from FILE")
        parser.add_option("-R", "--rx-subdev-spec", type="subdev",
default=None,
                          help="select USRP Rx side A or B (default=A)")
        parser.add_option("-f", "--freq", type="eng_float", default=100.1e6,
                          help="set frequency to FREQ", metavar="FREQ")
        parser.add_option("-g", "--gain", type="eng_float", default=40,
                          help="set gain in dB (default is midpoint)")
        parser.add_option("-V", "--volume", type="eng_float", default=None,
                          help="set volume (default is midpoint)")
      parser.add_option("-R", "--repeat", action="store_true", default=False)
        parser.add_option("-O", "--audio-output", type="string", default="",
                          help="pcm device name.  E.g., hw:0,0 or surround51
or /dev/dsp")

        (options, args) = parser.parse_args()
        if len(args) != 1:
            parser.print_help()
```

```
            sys.exit(1)

        self.frame = frame
        self.panel = panel

        self.vol = 0
        self.state = "FREQ"
        self.freq = 0

        # build graph

      if options.filename is USRP:
              self.u = usrp.source_c()                    # usrp is data source
        else:
            self.u = gr.file_source (gr.sizeof_float, options.filename,
options.repeat)

        adc_rate = self.u.adc_rate()                # 64 MS/s
        usrp_decim = 200
        self.u.set_decim_rate(usrp_decim)
        usrp_rate = adc_rate / usrp_decim           # 320 kS/s
        chanfilt_decim = 1
        demod_rate = usrp_rate / chanfilt_decim
        audio_decimation = 10
        audio_rate = demod_rate / audio_decimation  # 32 kHz

        if options.rx_subdev_spec is None:
            options.rx_subdev_spec = pick_subdevice(self.u)

        self.u.set_mux(usrp.determine_rx_mux_value(self.u,
options.rx_subdev_spec))
        self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
        print "Using RX d'board %s" % (self.subdev.side_and_name(),)


        chan_filt_coeffs = optfir.low_pass (1,           # gain
                                            usrp_rate,   # sampling rate
                                            80e3,        # passband cutoff
                                            115e3,       # stopband cutoff
                                            0.1,         # passband ripple
                                            60)          # stopband
attenuation
        #print len(chan_filt_coeffs)
        chan_filt = gr.fir_filter_ccf (chanfilt_decim, chan_filt_coeffs)

        self.guts = blks.wfm_rcv (self, demod_rate, audio_decimation)

        self.volume_control = gr.multiply_const_ff(self.vol)

      # file as final sink
        filename = args[0]
      file_sink = gr.file_sink (gr.sizeof_float, filename)

        # now wire it all together
        self.connect (self.u, chan_filt, self.guts, self.volume_control,
file_sink)
```

```
        self._build_gui(vbox, usrp_rate, demod_rate, audio_rate)

        if options.gain is None:
            # if no gain was specified, use the mid-point in dB
            g = self.subdev.gain_range()
            options.gain = float(g[0]+g[1])/2

        if options.volume is None:
            g = self.volume_range()
            options.volume = float(g[0]+g[1])/2

        if abs(options.freq) < 1e6:
            options.freq *= 1e6

        # set initial values

        self.set_gain(options.gain)
        self.set_vol(options.volume)
        if not(self.set_freq(options.freq)):
            self._set_status_msg("Failed to set initial frequency")


    def _set_status_msg(self, msg, which=0):
        self.frame.GetStatusBar().SetStatusText(msg, which)


    def _build_gui(self, vbox, usrp_rate, demod_rate, audio_rate):

        def _form_set_freq(kv):
            return self.set_freq(kv['freq'])


        if 1:
            self.src_fft = fftsink.fft_sink_c (self, self.panel, title="Data
from USRP",
                                               fft_size=512,
sample_rate=usrp_rate)
            self.connect (self.u, self.src_fft)
            vbox.Add (self.src_fft.win, 4, wx.EXPAND)

        if 1:
            post_filt_fft = fftsink.fft_sink_f (self, self.panel, title="Post
Demod",
                                                fft_size=1024,
sample_rate=usrp_rate,
                                                y_per_div=10, ref_level=0)
            self.connect (self.guts.fm_demod, post_filt_fft)
            vbox.Add (post_filt_fft.win, 4, wx.EXPAND)

        if 0:
            post_deemph_fft = fftsink.fft_sink_f (self, self.panel,
title="Post Deemph",
                                                  fft_size=512,
sample_rate=audio_rate,
                                                  y_per_div=10, ref_level=-
20)
            self.connect (self.guts.deemph, post_deemph_fft)
```

```
            vbox.Add (post_deemph_fft.win, 4, wx.EXPAND)


        # control area form at bottom
        self.myform = myform = form.form()

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox.Add((5,0), 0)
        myform['freq'] = form.float_field(
            parent=self.panel, sizer=hbox, label="Freq", weight=1,
            callback=myform.check_input_and_call(_form_set_freq,
self._set_status_msg))

        hbox.Add((5,0), 0)
        myform['freq_slider'] = \
            form.quantized_slider_field(parent=self.panel, sizer=hbox,
weight=3,
                                        range=(87.9e6, 108.1e6, 0.1e6),
                                        callback=self.set_freq)
        hbox.Add((5,0), 0)
        vbox.Add(hbox, 0, wx.EXPAND)

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox.Add((5,0), 0)

        myform['volume'] = \
            form.quantized_slider_field(parent=self.panel, sizer=hbox,
label="Volume",
                                        weight=3, range=self.volume_range(),
                                        callback=self.set_vol)
        hbox.Add((5,0), 1)

        myform['gain'] = \
            form.quantized_slider_field(parent=self.panel, sizer=hbox,
label="Gain",
                                        weight=3,
range=self.subdev.gain_range(),
                                        callback=self.set_gain)
        hbox.Add((5,0), 0)
        vbox.Add(hbox, 0, wx.EXPAND)

        try:
            self.knob = powermate.powermate(self.frame)
            self.rot = 0
            powermate.EVT_POWERMATE_ROTATE (self.frame, self.on_rotate)
            powermate.EVT_POWERMATE_BUTTON (self.frame, self.on_button)
        except:
            print "FYI: No Powermate or Contour Knob found"


    def on_rotate (self, event):
        self.rot += event.delta
        if (self.state == "FREQ"):
            if self.rot >= 3:
                self.set_freq(self.freq + .1e6)
                self.rot -= 3
            elif self.rot <=-3:
```

```
                self.set_freq(self.freq - .1e6)
                self.rot += 3
        else:
            step = self.volume_range()[2]
            if self.rot >= 3:
                self.set_vol(self.vol + step)
                self.rot -= 3
            elif self.rot <=-3:
                self.set_vol(self.vol - step)
                self.rot += 3

    def on_button (self, event):
        if event.value == 0:        # button up
            return
        self.rot = 0
        if self.state == "FREQ":
            self.state = "VOL"
        else:
            self.state = "FREQ"
        self.update_status_bar ()


    def set_vol (self, vol):
        g = self.volume_range()
        self.vol = max(g[0], min(g[1], vol))
        self.volume_control.set_k(10**(self.vol/10))
        self.myform['volume'].set_value(self.vol)
        self.update_status_bar ()

    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital down converter.
        """
        r = usrp.tune(self.u, 0, self.subdev, target_freq)

        if r:
            self.freq = target_freq
            self.myform['freq'].set_value(target_freq)         # update
displayed value
            self.myform['freq_slider'].set_value(target_freq)  # update
displayed value
            self.update_status_bar()
            self._set_status_msg("OK", 0)
            return True

        self._set_status_msg("Failed", 0)
        return False

    def set_gain(self, gain):
```

```
        self.myform['gain'].set_value(gain)     # update displayed value
        self.subdev.set_gain(gain)

    def update_status_bar (self):
        msg = "Volume:%r  Setting:%s" % (self.vol, self.state)
        self._set_status_msg(msg, 1)
        self.src_fft.set_baseband_freq(self.freq)

    def volume_range(self):
        return (-20.0, 0.0, 0.5)


if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "USRP WFM RX")
    app.MainLoop ()
```

Packet-based File transmission attempt:

file_reciever.py:

```
#!/usr/bin/env python
#
# Copyright 2005,2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import struct
import sys

# from current dir
from receive_path import receive_path
```

```
import fusb_options

#import os
#print os.getpid()
#raw_input('Attach and press enter: ')


class my_graph(gr.flow_graph):

    def __init__(self, demod_class, rx_callback, options):
        gr.flow_graph.__init__(self)
        self.rxpath = receive_path(self, demod_class, rx_callback, options)
        self.rxpath.subdev.select_rx_antenna('RX2')

#
////////////////////////////////////////////////////////////////////////////
#                                     main
#
////////////////////////////////////////////////////////////////////////////

global n_rcvd, n_right

def main():
    global n_rcvd, n_right

    n_rcvd = 0
    n_right = 0

    sink_file = open("./received_file.dat", 'a')

    def rx_callback(ok, payload):
        global n_rcvd, n_right
        (pktno,) = struct.unpack('!H', payload[0:2])
        n_rcvd += 1
        if ok:
            sink_file.write(payload[2:])
            n_right += 1

        print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
            ok, pktno, n_rcvd, n_right)



    demods = modulation_utils.type_1_demods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option,
conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice",
choices=demods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(demods.keys()),))

    receive_path.add_options(parser, expert_grp)
```

```
    for mod in demods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    # build the graph
    fg = my_graph(demods[options.modulation], rx_callback, options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    fg.start()        # start flow graph
    fg.wait()         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

file_transmitter.py:

```
#!/usr/bin/env python
#
# Copyright 2005,2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
```

```
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import struct
import sys
import os
import time

# from current dir
from receive_path import receive_path
from transmit_path import transmit_path
import fusb_options

#import os
#print os.getpid()
#raw_input('Attach and press enter: ')


class my_graph(gr.flow_graph):

    def __init__(self, demod_class, modulator_class, rx_callback, options):
        gr.flow_graph.__init__(self)
        self.rxpath = receive_path(self, demod_class, rx_callback, options)
        self.rxpath.subdev.select_rx_antenna('RX2')
        self.txpath = transmit_path(self, modulator_class, options)


#
///////////////////////////////////////////////////////////////////////////
#                                   main
#
///////////////////////////////////////////////////////////////////////////

global n_rcvd, n_right, pktnot

def main():
    global n_rcvd, n_right, pktnot

    n_rcvd = 0
    n_right = 0

    pktnot = 0

    def send_pkt(payload='', eof=False):
        return fg.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        global n_rcvd, n_right, pktnot
        sink_file = open("./received_file.dat", 'a')
        (pktno,) = struct.unpack('!H', payload[0:2])
```

```
        n_rcvd += 1
        if ok:
            if n_right % 5 is not 0:
                  sink_file.write(payload[2:])
                  #n_right += 1
            if n_right % 5 == 0:
                # generate and send packets
                time.sleep(1)
                source_file = open("./received_file.dat", 'r')
                nbytes = int(1e6 * 2.5)
                n = 0

                pkt_size = int(options.size)

                while n < nbytes:

                    data = source_file.read(pkt_size - 2)

                    if data == '':
                        break;

                    payload = struct.pack('!H', pktnot) + data

                    send_pkt(payload)

                    n += len(payload)

                    sys.stderr.write('.')


                    pktnot += 1

                os.remove("./received_file.dat")
            n_right += 1


        print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
            ok, pktno, n_rcvd, n_right)



    demods = modulation_utils.type_1_demods()
    mods = modulation_utils.type_1_mods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option,
conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-q", "--demodulation", type="choice",
choices=demods.keys(),
                      default='gmsk',
                      help="Select demodulation from: %s [default=%%default]"
                            % (', '.join(demods.keys()),))
    parser.add_option("-m", "--modulation", type="choice",
choices=mods.keys(),
                      default='gmsk',
```

```
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(mods.keys()),))
    parser.add_option("-s", "--size", type="eng_float", default=1500,
                      help="set packet size [default=%default]")
    parser.add_option("-T", "--tx-subdev-spec", type="subdev", default=None,
                          help="select USRP Tx side A or B")
    parser.add_option("", "--tx-freq", type="eng_float", default=None,
                          help="set transmit frequency to FREQ
[default=%default]", metavar="FREQ")
    parser.add_option("-i", "--interp", type="intx", default=None,
                          help="set fpga interpolation rate to INTERP
[default=%default]")

    parser.add_option("", "--tx-amplitude", type="eng_float", default=12000,
metavar="AMPL",
                          help="set transmitter digital amplitude: 0 <= AMPL
< 32768 [default=%default]")

    receive_path.add_options(parser, expert_grp)


    for mod in demods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    # build the graph
    fg = my_graph(demods[options.modulation], mods[options.modulation],
rx_callback, options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    fg.start()        # start flow graph
    fg.wait()         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

fpga_file_transmitter.py:

```
#!/usr/bin/env python
#
# Copyright 2005, 2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random, time, struct, sys

# from current dir
from transmit_path import transmit_path
import fusb_options

#import os
#print os.getpid()
#raw_input('Attach and press enter')


class my_graph(gr.flow_graph):
    def __init__(self, modulator_class, options):
        gr.flow_graph.__init__(self)
        sample_rate = 32000
      ampl = 0.1

      src = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)

      filename = "/dev/FPGA_in"
      dst = gr.file_sink (gr.sizeof_float, filename)
        head = gr.head(gr.sizeof_float, 250000)
        self.connect((src, 0), head, dst)
      self.txpath = transmit_path(self, modulator_class, options)
```

```
#
/////////////////////////////////////////////////////////////////////////////
#                                     main
#
/////////////////////////////////////////////////////////////////////////////

def main():

    def send_pkt(payload='', eof=False):
        return fg.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        print "ok = %r, payload = '%s'" % (ok, payload)

    mods = modulation_utils.type_1_mods()

    parser = OptionParser(option_class=eng_option,
conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice",
choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(mods.keys()),))

    parser.add_option("-s", "--size", type="eng_float", default=500,
                      help="set packet size [default=%default]")
    parser.add_option("-b", "--burst-size", type="eng_float", default=500,
                      help="set packet burst size [default=%default]")
    parser.add_option("-M", "--megabytes", type="eng_float", default=3.0,
                      help="set megabytes to transmit [default=%default]")
    parser.add_option("","--discontinuous", action="store_true",
default=False,
                      help="enable discontinous transmission (bursts of
packets determined by burst size)")


    transmit_path.add_options(parser, expert_grp)

    for mod in mods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        sys.exit(1)

    if options.tx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)
```

```
    source_file = open("/dev/FPGA_out", 'r')

    # build the graph

    fg = my_graph(mods[options.modulation], options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: failed to enable realtime scheduling"

    fg.start()                       # start flow graph

    # generate and send packets
    nbytes = int(1e6 * options.megabytes)
    n = 0
    pktno = 0
    pkt_size = int(options.size)

    while n < nbytes:
        if options.file_source is None:
            data = (pkt_size - 2) * chr(pktno & 0xff)

        else:
            data = source_file.read(pkt_size - 2)
            if data == '':
                break;

        payload = struct.pack('!H', pktno) + data

        send_pkt(payload)

        n += len(payload)

        sys.stderr.write('.')

        if options.discontinuous and pktno % options.burst_size ==
(options.burst_size - 1):
            time.sleep(5)
        pktno += 1

    print 'pktno = ', pktno, 'n = ', n
    raw_input('Press any key to continue.')
    send_pkt(eof=True)
    fg.wait()                        # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

loopback_demo.py:

```
#!/usr/bin/env python
#
# Copyright 2005,2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import struct
import sys
import os
import time

# from current dir
from receive_path import receive_path
from transmit_path import transmit_path
import fusb_options

#import os
#print os.getpid()
#raw_input('Attach and press enter: ')


class my_graph(gr.flow_graph):

    def __init__(self, demod_class, modulator_class, rx_callback, options):
        gr.flow_graph.__init__(self)
        self.rxpath = receive_path(self, demod_class, rx_callback, options)
        self.rxpath.subdev.select_rx_antenna('RX2')
        self.txpath = transmit_path(self, modulator_class, options)


#
///////////////////////////////////////////////////////////////////////////
#                                   main
```

```
#
/////////////////////////////////////////////////////////////////////////////

global n_rcvd, n_right, pktnot

def main():
    global n_rcvd, n_right, pktnot

    n_rcvd = 0
    n_right = 0

    pktnot = 0

    def send_pkt(payload='', eof=False):
        return fg.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        global n_rcvd, n_right, pktnot
        sink_file = open("./received_file.dat", 'a')
        (pktno,) = struct.unpack('!H', payload[0:2])
        n_rcvd += 1
        if ok:
            if n_right % 5 is not 0:
                  sink_file.write(payload[2:])
                  #n_right += 1
            if n_right % 5 == 0:
                # generate and send packets
                time.sleep(1)
                source_file = open("./received_file.dat", 'r')
                nbytes = int(1e6 * 2.5)
                n = 0

                pkt_size = int(options.size)

                while n < nbytes:

                    data = source_file.read(pkt_size - 2)

                    if data == '':
                        break;

                    payload = struct.pack('!H', pktnot) + data

                    send_pkt(payload)

                    n += len(payload)

                    sys.stderr.write('.')


                    pktnot += 1

                os.remove("./received_file.dat")
            n_right += 1


        print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
```

```
            ok, pktno, n_rcvd, n_right)


    demods = modulation_utils.type_1_demods()
    mods = modulation_utils.type_1_mods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option,
conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-q", "--demodulation", type="choice",
choices=demods.keys(),
                      default='gmsk',
                      help="Select demodulation from: %s [default=%%default]"
                            % (', '.join(demods.keys()),))
    parser.add_option("-m", "--modulation", type="choice",
choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(mods.keys()),))
    parser.add_option("-s", "--size", type="eng_float", default=1500,
                      help="set packet size [default=%default]")
    parser.add_option("-T", "--tx-subdev-spec", type="subdev", default=None,
                          help="select USRP Tx side A or B")
    parser.add_option("", "--tx-freq", type="eng_float", default=None,
                          help="set transmit frequency to FREQ
[default=%default]", metavar="FREQ")
    parser.add_option("-i", "--interp", type="intx", default=None,
                          help="set fpga interpolation rate to INTERP
[default=%default]")

    parser.add_option("", "--tx-amplitude", type="eng_float", default=12000,
metavar="AMPL",
                          help="set transmitter digital amplitude: 0 <= AMPL
< 32768 [default=%default]")

    receive_path.add_options(parser, expert_grp)


    for mod in demods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)
```

```
    # build the graph
    fg = my_graph(demods[options.modulation], mods[options.modulation],
rx_callback, options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    fg.start()        # start flow graph
    fg.wait()         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Packet-based message transmission:

nofpga_message_transmitter.py:

```
#!/usr/bin/env python
#
# Copyright 2005, 2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random, time, struct, sys

# from current dir
from transmit_path import transmit_path
import fusb_options
```

```
#import os
#print os.getpid()
#raw_input('Attach and press enter')


class my_graph(gr.flow_graph):
    def __init__(self, modulator_class, options):
        gr.flow_graph.__init__(self)
        self.txpath = transmit_path(self, modulator_class, options)




#
///////////////////////////////////////////////////////////////////////////
#                                     main
#
///////////////////////////////////////////////////////////////////////////

def main():

    def send_pkt(payload='', eof=False):
        return fg.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        print "ok = %r, payload = '%s'" % (ok, payload)

    mods = modulation_utils.type_1_mods()

    parser = OptionParser(option_class=eng_option,
conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice",
choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(mods.keys()),))

    transmit_path.add_options(parser, expert_grp)

    for mod in mods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        sys.exit(1)

    if options.tx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)
```

```
    # build the graph

    fg = my_graph(mods[options.modulation], options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: failed to enable realtime scheduling"

    fg.start()                       # start flow graph

    # generate and send packets
    pktno = 0
    data = raw_input("Enter your message here: ")

    payload = struct.pack('!H', pktno) + data

    send_pkt(payload)

    sys.stderr.write('Message Transmission Complete. ')

    raw_input('Press any key to continue.')
    send_pkt(eof=True)
    fg.wait()                        # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

fpga_message_transmitter.py;

```
#!/usr/bin/env python
#
# Copyright 2005, 2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
```

```
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random, time, struct, sys

# from current dir
from transmit_path import transmit_path
import fusb_options

#import os
#print os.getpid()
#raw_input('Attach and press enter')


class my_graph(gr.flow_graph):
    def __init__(self, modulator_class, options):
        gr.flow_graph.__init__(self)

      src = gr.file_source (gr.sizeof_char, "./message.dat")

      filename = "/dev/FPGA_in"
      dst = gr.file_sink (gr.sizeof_float, filename)
       self.connect((src, 0), dst)
      self.txpath = transmit_path(self, modulator_class, options)




#
////////////////////////////////////////////////////////////////////////////
#                                     main
#
////////////////////////////////////////////////////////////////////////////

def main():

    def send_pkt(payload='', eof=False):
        return fg.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        print "ok = %r, payload = '%s'" % (ok, payload)

    mods = modulation_utils.type_1_mods()

    parser = OptionParser(option_class=eng_option,
conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice",
choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
```

```
                            % (', '.join(mods.keys()),))

    transmit_path.add_options(parser, expert_grp)

    for mod in mods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        sys.exit(1)

    if options.tx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    source_file = open("/dev/FPGA_out", 'r')

    # build the graph

    fg = my_graph(mods[options.modulation], options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: failed to enable realtime scheduling"

    fg.start()                        # start flow graph

    # generate and send packets

    pktno = 0
    pkt_size = int(options.size)
    message = "hello world" # need to find a better way to determine file
length eventually
    data = source_file.read(len(message))

    payload = struct.pack('!H', pktno) + data

    send_pkt(payload)

    sys.stderr.write('Message Transmitted.')

    pktno += 1

    print 'pktno = ', pktno
    raw_input('Press any key to continue.')
    send_pkt(eof=True)
    fg.wait()                         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
```

```
        pass
```

relay_small.py:

```
#!/usr/bin/env python
#
# Copyright 2005,2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import struct
import sys
import os
import time

# from current dir
from receive_path import receive_path
from transmit_path import transmit_path
import fusb_options

#import os
#print os.getpid()
#raw_input('Attach and press enter: ')


class my_graph(gr.flow_graph):

    def __init__(self, demod_class, modulator_class, rx_callback, options):
        gr.flow_graph.__init__(self)
        self.rxpath = receive_path(self, demod_class, rx_callback, options)
        self.rxpath.subdev.select_rx_antenna('RX2')
        self.txpath = transmit_path(self, modulator_class, options)


# ////////////////////////////////////////////////////////////////////
#                                   main
# ////////////////////////////////////////////////////////////////////
```

```
global n_rcvd, n_right, pktnot

def main():
    global n_rcvd, n_right, pktnot
    n_rcvd = 0
    n_right = 0
    pktnot = 0
    def send_pkt(payload='', eof=False):
        return fg.txpath.send_pkt(payload, eof)
    def rx_callback(ok, payload):
        global n_rcvd, n_right, pktnot

        (pktno,) = struct.unpack('!H', payload[0:2])
        n_rcvd += 1
        if ok:

            print payload[2:]


            # generate and send packets
            time.sleep(1)

            pkt_size = len(payload)

            data = payload[2:]

            payload2 = struct.pack('!H', pktnot) + data

            send_pkt(payload2)

            sys.stderr.write('Message Received and Relayed.')


            pktnot += 1
            n_right += 1
        if not ok:
            print "Packet not received."

        print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (ok, pktno,
n_rcvd, n_right)



    demods = modulation_utils.type_1_demods()
    mods = modulation_utils.type_1_mods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option, conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-q", "--demodulation", type="choice", choices=demods.keys(),
                      default='gmsk',
                      help="Select demodulation from: %s [default=%%default]"
                            % (', '.join(demods.keys()),))
    parser.add_option("-m", "--modulation", type="choice", choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(mods.keys()),))
    parser.add_option("-s", "--size", type="eng_float", default=1500,
                      help="set packet size [default=%default]")
    parser.add_option("-b", "--burst-size", type="eng_float", default=500,
                      help="set packet burst size [default=%default]")
    parser.add_option("-T", "--tx-subdev-spec", type="subdev", default=None,
```

```
                          help="select USRP Tx side A or B")
    parser.add_option("", "--tx-freq", type="eng_float", default=None,
                          help="set transmit frequency to FREQ [default=%default]",
metavar="FREQ")
    parser.add_option("-i", "--interp", type="intx", default=None,
                          help="set fpga interpolation rate to INTERP
[default=%default]")

    parser.add_option("", "--tx-amplitude", type="eng_float", default=12000,
metavar="AMPL",
                          help="set transmitter digital amplitude: 0 <= AMPL < 32768
[default=%default]")

    receive_path.add_options(parser, expert_grp)


    for mod in demods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    # build the graph
    fg = my_graph(demods[options.modulation], mods[options.modulation], rx_callback,
options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    fg.start()        # start flow graph
    fg.wait()         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

message_receiver.py:

```
#!/usr/bin/env python
#
# Copyright 2005,2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
```

```
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import struct
import sys

# from current dir
from receive_path import receive_path
import fusb_options

#import os
#print os.getpid()
#raw_input('Attach and press enter: ')


class my_graph(gr.flow_graph):

    def __init__(self, demod_class, rx_callback, options):
        gr.flow_graph.__init__(self)
        self.rxpath = receive_path(self, demod_class, rx_callback, options)
        self.rxpath.subdev.select_rx_antenna('RX2')

#
////////////////////////////////////////////////////////////////////////////
#                                     main
#
////////////////////////////////////////////////////////////////////////////

global n_rcvd, n_right

def main():
    global n_rcvd, n_right

    n_rcvd = 0
    n_right = 0
```

```
    def rx_callback(ok, payload):
        global n_rcvd, n_right
        sink_file = open("./received_file.dat", 'a')
        (pktno,) = struct.unpack('!H', payload[0:2])
        n_rcvd += 1
        if ok:
            sink_file.write(payload[2:])
            print "Incoming message: ", payload[2:]
            n_right += 1
        sink_file.close()

        #print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
            #ok, pktno, n_rcvd, n_right)



    demods = modulation_utils.type_1_demods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option,
conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice",
choices=demods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(demods.keys()),))

    receive_path.add_options(parser, expert_grp)


    for mod in demods.values():
        mod.add_options(expert_grp)

    fusb_options.add_options(expert_grp)
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    # build the graph
    fg = my_graph(demods[options.modulation], rx_callback, options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    fg.start()        # start flow graph
    fg.wait()         # wait for it to finish
```

```
if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Stream-based File Transmission:

streaming_loopback.py:

```
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru
from gnuradio import usrp
from gnuradio import audio
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
import wx

class my_graph(gr.flow_graph):

    def __init__(self):
      gr.flow_graph.__init__(self)

            self.source = usrp.source_c(0, 250)
      self.sink = usrp.sink_c (0, 250)
        self.connect(self.source, self.sink)

    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
```

```
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """
      r = self.sink.tune(self.subdev._which, self.subdev, target_freq)
            if r:
                  #print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
                  #print "r.dxc_freq      =",
eng_notation.num_to_str(r.dxc_freq)
                  #print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
                  #print "r.inverted      =", r.inverted
                  return True

            return False

def main ():
    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-T", "--tx-subdev-spec", type="subdev", default=(0,
0),
                       help="select USRP Tx side A or B")
    parser.add_option ("-f", "--rf-freq", type="eng_float", default=None,
                       help="set RF center frequency to FREQ")
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        raise SystemExit

    if options.rf_freq is None:
        sys.stderr.write("usrp: must specify RF center frequency with -f
RF_FREQ\n")
        parser.print_help()
        raise SystemExit

    fg = my_graph()

    # determine the daughterboard subdevice we're using
    if options.tx_subdev_spec is None:
        options.tx_subdev_spec = usrp.pick_tx_subdevice(fg.u)

    m = usrp.determine_tx_mux_value(fg.sink, options.tx_subdev_spec)
    #print "mux = %#04x" % (m,)
    fg.sink.set_mux(m)
    fg.subdev = usrp.selected_subdev(fg.sink, options.tx_subdev_spec)
    print "Using TX d'board %s" % (fg.subdev.side_and_name(),)

    fg.subdev.set_gain(fg.subdev.gain_range()[1])    # set max Tx gain

    if not fg.set_freq(options.rf_freq):
        sys.stderr.write('Failed to set RF frequency\n')
        raise SystemExit

    fg.subdev.set_enable(True)                       # enable transmitter

    try:
```

```
        fg.run()
    except KeyboardInterrupt:
        pass

if __name__ == '__main__':
      main()
```

streaming_loopback_receiver.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import usrp_dbid
import sys
import math

def pick_subdevice(u):
    """
    The user didn't specify a subdevice on the command line.
    Try for one of these, in order: TV_RX, BASIC_RX, whatever is on side A.

    @return a subdev_spec
    """
    return usrp.pick_subdev(u, (usrp_dbid.TV_RX,
                                usrp_dbid.TV_RX_REV_2,
                                usrp_dbid.BASIC_RX))



class my_graph (gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)

        parser=OptionParser(option_class=eng_option)
        parser.add_option("-R", "--rx-subdev-spec", type="subdev",
default=None,
                          help="select USRP Rx side A or B (default=A)")
        parser.add_option("-f", "--freq", type="eng_float", default=100.1e6,
                          help="set frequency to FREQ", metavar="FREQ")
        parser.add_option("-g", "--gain", type="eng_float", default=None,
                          help="set gain in dB (default is midpoint)")
      parser.add_option ("-T", "--tx-subdev-spec", type="subdev", default=(0,
0),
                         help="select USRP Tx side A or B")


        (options, args) = parser.parse_args()
        if len(args) != 0:
            parser.print_help()
            sys.exit(1)
```

```
        self.vol = .1
        self.state = "FREQ"
        self.freq = 0

        # build graph

        self.u = usrp.source_c(0, 250)                    # usrp is data
source

        adc_rate = self.u.adc_rate()                # 64 MS/s
        usrp_decim = 250
        self.u.set_decim_rate(usrp_decim)
        usrp_rate = adc_rate / usrp_decim           # 320 kS/s
        chanfilt_decim = 1

        if options.rx_subdev_spec is None:
            options.rx_subdev_spec = pick_subdevice(self.u)

        self.u.set_mux(usrp.determine_rx_mux_value(self.u,
options.rx_subdev_spec))
        self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
        print "Using RX d'board %s" % (self.subdev.side_and_name(),)


        chan_filt_coeffs = optfir.low_pass (1,           # gain
                                            usrp_rate,   # sampling rate
                                            80e3,        # passband cutoff
                                            115e3,       # stopband cutoff
                                            0.1,         # passband ripple
                                            60)          # stopband
attenuation
        #print len(chan_filt_coeffs)
        chan_filt = gr.fir_filter_ccf (chanfilt_decim, chan_filt_coeffs)

        self.demod = blks.dbpsk_demod(self,
                                  2, #samples per symbol
                                  .35, #excess bandwidth
                                  .1, #alpha
                                  None, #gain
                                  0.5, #mu
                                  .005, #omega relative limit
                                  True, #gray code
                                  False, #verbose mode
                                  False) #logging

        self.volume_control = gr.multiply_const_ff(self.vol)

        # file sinks and soruces
        dst = gr.file_sink (1, "received_file.dat")

      src = gr.file_source (gr.sizeof_float, "received_file.dat")

      ampdst = gr.file_sink (gr.sizeof_float, "amplified_received_file.dat")
        # now wire it all together
        self.connect (self.u, chan_filt)
      self.connect (chan_filt, self.demod)
```

```
  self.connect (self.demod, dst)
  self.connect (src, self.volume_control, ampdst)


    if options.gain is None:
        # if no gain was specified, use the mid-point in dB
        g = self.subdev.gain_range()
        options.gain = float(g[0]+g[1])/2

    if abs(options.freq) < 1e6:
        options.freq *= 1e6

    # set initial values

    self.set_gain(options.gain)

    if not(self.set_freq(options.freq)):
        self._set_status_msg("Failed to set initial frequency")

def set_vol (self, vol):
    self.vol = vol
    self.volume_control.set_k(self.vol)
    self.update_status_bar ()

def set_freq(self, target_freq):
    """
    Set the center frequency we're interested in.

    @param target_freq: frequency in Hz
    @rypte: bool

    Tuning is a two step process.  First we ask the front-end to
    tune as close to the desired frequency as it can.  Then we use
    the result of that operation and our target_frequency to
    determine the value for the digital down converter.
    """
    r = self.u.tune(0, self.subdev, target_freq)

    if r:
        self.freq = target_freq
        self.update_status_bar()
        self._set_status_msg("OK", 0)
        return True

    self._set_status_msg("Failed", 0)
    return False

def set_gain(self, gain):
    self.subdev.set_gain(gain)

def update_status_bar (self):
    msg = "Freq: %s  Volume:%f  Setting:%s" % (
        eng_notation.num_to_str(self.freq), self.vol, self.state)
    self._set_status_msg(msg, 1)

def _set_status_msg(self, msg, which=0):
    print msg
```

```
if __name__ == '__main__':
    fg = my_graph()
    try:
        fg.run()
    except KeyboardInterrupt:
        pass
```

streaming_loopback_transmitter.py:

```
#!/usr/bin/env python
#
# Copyright 2004 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru
from gnuradio import usrp
from gnuradio import audio
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
import wx

class my_graph(gr.flow_graph):

    def __init__(self):
      gr.flow_graph.__init__(self)

      sample_rate = int(32000)
      ampl = 0.1

      self.signal = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
      self.modulation = blks.dbpsk_mod(self,
                                       2, #samples per symbol
                                            .35, #excess bandwidth
                                            True, #gray code?
```

```
                                        False, #verbose?
                                        False) #logging?
      self.fsink = gr.file_sink (gr.sizeof_float, "transmitted_file.dat")
      self.fsource = gr.file_source (gr.sizeof_char, "transmitted_file.dat")
      self.head = gr.head(gr.sizeof_float, int(50000))
      self.sink = usrp.sink_c (0, 200)
      self.connect((self.signal, 0), self.head, self.fsink)
      self.connect(self.fsource, self.modulation, self.sink)

    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """
      r = self.sink.tune(self.subdev._which, self.subdev, target_freq)
            if r:
                  #print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
                  #print "r.dxc_freq      =",
eng_notation.num_to_str(r.dxc_freq)
                  #print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
                  #print "r.inverted      =", r.inverted
                  return True

            return False

def main ():

    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-T", "--tx-subdev-spec", type="subdev", default=(0,
0),
                       help="select USRP Tx side A or B")
    parser.add_option ("-f", "--rf-freq", type="eng_float", default=None,
                       help="set RF center frequency to FREQ")
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        raise SystemExit

    if options.rf_freq is None:
        sys.stderr.write("usrp: must specify RF center frequency with -f
RF_FREQ\n")
        parser.print_help()
        raise SystemExit

    fg = my_graph()
```

```
    # determine the daughterboard subdevice we're using
    if options.tx_subdev_spec is None:
        options.tx_subdev_spec = usrp.pick_tx_subdevice(fg.u)

    m = usrp.determine_tx_mux_value(fg.sink, options.tx_subdev_spec)
    #print "mux = %#04x" % (m,)
    fg.sink.set_mux(m)
    fg.subdev = usrp.selected_subdev(fg.sink, options.tx_subdev_spec)
    print "Using TX d'board %s" % (fg.subdev.side_and_name(),)

    fg.subdev.set_gain(fg.subdev.gain_range()[1])    # set max Tx gain

    if not fg.set_freq(options.rf_freq):
        sys.stderr.write('Failed to set RF frequency\n')
        raise SystemExit

    fg.subdev.set_enable(True)                       # enable transmitter

    try:
        fg.run()
    except KeyboardInterrupt:
        pass

if __name__ == '__main__':
      main()
```

TCP/IP File Transmission:

TCPIP_file_transmitter.py:

```
#!/usr/bin/env python
#
# Copyright 2004 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_option import eng_option
```

```
from optparse import OptionParser

import sys
import os

def send_A_file():

      os.system("tar cf - ./transmitted_signal.dat | ssh frank@192.168.200.1
'/home/frank/tunnelback.sh'")

class my_graph(gr.flow_graph):

    def __init__(self):
      gr.flow_graph.__init__(self)

      usage="%prog: [options]"
      parser = OptionParser(option_class=eng_option, usage=usage)
      parser.add_option("-r", "--sample-rate", type="eng_float",
default=48000,
                        help="set sample rate to RATE (48000)")
      parser.add_option("-N", "--nsamples", type="eng_float", default=None,
                        help="number of samples to collect [default=+inf]")

      (options, args) = parser.parse_args ()
      if len(args) != 0:
            parser.print_help()
            raise SystemExit, 1

      sample_rate = int(options.sample_rate)
      ampl = 0.1

      src = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)

      dst = gr.file_sink (gr.sizeof_float, "./transmitted_signal.dat")

      if options.nsamples is None:
            self.connect((src, 0), dst)
        else:
            head = gr.head(gr.sizeof_float, int(options.nsamples))
            self.connect((src, 0), head, dst)

if __name__ == '__main__':
    try:
        my_graph().run()
        send_A_file()
    except KeyboardInterrupt:
        pass
```

tunnelback.sh:

```
#!/bin/sh

ssh fbivona@192.168.200.2 'cd ~/received_files && tar xpvf -'
```

Signal Processing Block:

usrp_randsiggen.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru
from gnuradio import randsig
from gnuradio import usrp
from gnuradio.eng_option import eng_option
from gnuradio import eng_notation
from optparse import OptionParser
import sys



class my_graph(gr.flow_graph):
    def __init__ (self):
        gr.flow_graph.__init__(self)

        # controllable values
        self.interp = 64
        self._instantiate_blocks ()

    def usb_freq (self):
        return self.u.dac_freq() / self.interp

    def usb_throughput (self):
        return self.usb_freq () * 4

    def set_interpolator (self, interp):
        self.interp = interp
        self.siggen.set_sampling_freq (self.usb_freq ())
        self.u.set_interp_rate (interp)

    def _instantiate_blocks (self):
        self.src = None
        self.u = usrp.sink_c (0, self.interp)

        self.siggen = randsig.source_ff (self.usb_freq ())

        # self.file_sink = gr.file_sink (gr.sizeof_gr_complex, "siggen.dat")

    def _configure_graph (self, type):
        was_running = self.is_running ()
        if was_running:
            self.stop ()
        self.disconnect_all ()
        self.connect (self.siggen, self.u)
        # self.connect (self.siggen, self.file_sink)
        self.src = self.siggen
        if was_running:
            self.start ()

    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.
```

```
        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """
        r = self.u.tune(self.subdev._which, self.subdev, target_freq)
        if r:
            #print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
            #print "r.dxc_freq      =", eng_notation.num_to_str(r.dxc_freq)
            #print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
            #print "r.inverted      =", r.inverted
            return True

        return False



def main ():
    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-T", "--tx-subdev-spec", type="subdev", default=(0,
0),
                       help="select USRP Tx side A or B")
    parser.add_option ("-f", "--rf-freq", type="eng_float", default=None,
                       help="set RF center frequency to FREQ")
    parser.add_option ("-i", "--interp", type="int", default=64,
                       help="set fgpa interpolation rate to INTERP
[default=%default]")

    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        raise SystemExit

    if options.rf_freq is None:
        sys.stderr.write("usrp_siggen: must specify RF center frequency with
-f RF_FREQ\n")
        parser.print_help()
        raise SystemExit

    fg = my_graph()
    fg.set_interpolator (options.interp)

    # determine the daughterboard subdevice we're using
    if options.tx_subdev_spec is None:
        options.tx_subdev_spec = usrp.pick_tx_subdevice(fg.u)

    m = usrp.determine_tx_mux_value(fg.u, options.tx_subdev_spec)
    #print "mux = %#04x" % (m,)
    fg.u.set_mux(m)
    fg.subdev = usrp.selected_subdev(fg.u, options.tx_subdev_spec)
```

```
    print "Using TX d'board %s" % (fg.subdev.side_and_name(),)

    fg.subdev.set_gain(fg.subdev.gain_range()[1])    # set max Tx gain

    if not fg.set_freq(options.rf_freq):
        sys.stderr.write('Failed to set RF frequency\n')
        raise SystemExit

    fg.subdev.set_enable(True)                       # enable transmitter

    try:
        fg.run()
    except KeyboardInterrupt:
        pass

if __name__ == '__main__':
    main ()
```

randsig_source_ff.h:

```
/* -*- c++ -*- */
/*
 * Copyright 2004 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

#ifndef INCLUDED_RANDSIG_SOURCE_FF_H
#define INCLUDED_RANDSIG_SOURCE_FF_H

#include <gr_sync_block.h>
#include <gr_fxpt_nco.h>

class randsig_source_ff;
typedef boost::shared_ptr<randsig_source_ff> randsig_source_ff_sptr;

/*!
 * \brief signal generator with float output.
```

```
 * \ingroup source
 */

class randsig_source_ff : public gr_sync_block {
  friend randsig_source_ff_sptr
  randsig_make_source_ff (double sampling_freq);

  double          d_sampling_freq; // parameter
  int             d_waveform; // determined psuedo-randomly
  long                  d_frequency; // determined psuedo-randomly
  long                  d_ampl; // determined psuedo-randomly
  float           d_offset; // determined psuedo-randomly
  gr_fxpt_nco           d_nco; // determined psuedo-randomly


  randsig_source_ff (double sampling_freq);

 public:
  virtual int work (int noutput_items,
                gr_vector_const_void_star &input_items,
                gr_vector_void_star &output_items);

  // ACCESSORS
  double sampling_freq () const { return d_sampling_freq; }
  int waveform () const { return d_waveform; }
  long frequency () const { return d_frequency; }
  long amplitude () const { return d_ampl; }
  float offset () const { return d_offset; }

  // MANIPULATORS
  void set_sampling_freq (double sampling_freq);
  void set_waveform (void);
  void set_frequency (void);
  void set_amplitude (void);
  void set_offset (float offset);
};

randsig_source_ff_sptr
randsig_make_source_ff (double sampling_freq);


#endif
```

randsig.i:

```
/* -*- c++ -*- */

%feature("autodoc", "1");           // generate python docstrings

%include "exception.i"
%import "gnuradio.i"                // the common stuff

%{
#include "gnuradio_swig_bug_workaround.h" // mandatory bug fix
#include "randsig_source_ff.h"
#include "randsig_source_ff.h"
```

```
#include <stdexcept>
%}

// ----------------------------------------------------------------

/*
 * First arg is the package prefix.
 * Second arg is the name of the class minus the prefix.
 *
 * This does some behind-the-scenes magic so we can
 * access howto_square_ff from python as howto.square_ff
 */
GR_SWIG_BLOCK_MAGIC(randsig,source_ff);

randsig_source_ff_sptr
randsig_make_source_ff (double sampling_freq);


class randsig_source_ff : public gr_sync_block {
 private:
  randsig_source_ff (double sampling_freq);

 public:

  // ACCESSORS
  double sampling_freq () const { return d_sampling_freq; }
  int waveform () const { return d_waveform; }
  long frequency () const { return d_frequency; }
  long amplitude () const { return d_ampl; }
  float offset () const { return d_offset; }

  // MANIPULATORS
  void set_sampling_freq (double sampling_freq);
  void set_waveform (void);
  void set_frequency (void);
  void set_amplitude (void);
  void set_offset (float offset);
};
```

randsig_source_ff.cc:

```
/* -*- c++ -*- */
/*
 * Copyright 2004 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
```

```
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
#include <randsig_source_ff.h>
#include <algorithm>
#include <gr_io_signature.h>
#include <stdexcept>
#include <gr_complex.h>


randsig_source_ff::randsig_source_ff (double sampling_freq)
  : gr_sync_block ("source_ff",
              gr_make_io_signature (0, 0, 0),
              gr_make_io_signature (1, 1, sizeof (float))),
    d_sampling_freq (sampling_freq), d_waveform (rand() % 6), d_frequency
(((rand() % 1000) + 100)*1000000),
    d_ampl ((rand() % 101) + 10), d_offset (0)
{
  d_nco.set_freq (2 * M_PI * d_frequency / d_sampling_freq);
}

randsig_source_ff_sptr
randsig_make_source_ff (double sampling_freq)
{
  return randsig_source_ff_sptr (new randsig_source_ff (sampling_freq));
}

int
randsig_source_ff::work (int noutput_items,
                gr_vector_const_void_star &input_items,
                gr_vector_void_star &output_items)
{
  float *optr = (float *) output_items[0];
  float t;

  switch (d_waveform){

  case 0: // Constant wave
    t = (float) d_ampl + d_offset;
    for (int i = 0; i < noutput_items; i++)     // FIXME unroll
      optr[i] = t;
    break;

  case 1: // Sine wave
    d_nco.sin (optr, noutput_items, d_ampl);
    if (d_offset == 0)
      break;
```

```
    for (int i = 0; i < noutput_items; i++){
      optr[i] += d_offset;
    }
    break;

  case 2: // Cosine wave
    d_nco.cos (optr, noutput_items, d_ampl);
    if (d_offset == 0)
      break;

    for (int i = 0; i < noutput_items; i++){
      optr[i] += d_offset;
    }
    break;

  /* The square wave is high from -PI to 0.     */
  case 3:
    t = (float) d_ampl + d_offset;
    for (int i = 0; i < noutput_items; i++){
      if (d_nco.get_phase() < 0)
        optr[i] = t;
      else
        optr[i] = d_offset;
      d_nco.step();
    }
    break;

  /* The triangle wave rises from -PI to 0 and falls from 0 to PI.      */
  case 4:
    for (int i = 0; i < noutput_items; i++){
      double t = d_ampl*d_nco.get_phase()/M_PI;
      if (d_nco.get_phase() < 0)
      optr[i] = static_cast<float>(t + d_ampl + d_offset);
      else
      optr[i] = static_cast<float>(-1*t + d_ampl + d_offset);
      d_nco.step();
    }
    break;

  /* The saw tooth wave rises from -PI to PI.   */
  case 5:
    for (int i = 0; i < noutput_items; i++){
      t = static_cast<float>(d_ampl*d_nco.get_phase()/(2*M_PI) + d_ampl/2 +
d_offset);
      optr[i] = t;
      d_nco.step();
    }
    break;

  default:
    throw std::runtime_error ("rand_sig_source: invalid waveform");
  }

  return noutput_items;
}

void
```

```
randsig_source_ff::set_sampling_freq (double sampling_freq)
{
  d_sampling_freq = sampling_freq;
  d_nco.set_freq (2 * M_PI * d_frequency / d_sampling_freq);
}

void
randsig_source_ff::set_waveform (void)
{
  d_waveform = rand() % 6;
}

void
randsig_source_ff::set_frequency (void)
{
  d_frequency = ((rand() % 1000) + 100)*1000000; // random number ranging
from 100M to 1G
  d_nco.set_freq (2 * M_PI * d_frequency / d_sampling_freq);
}

void
randsig_source_ff::set_amplitude (void)
{
  d_ampl = (rand() % 101) + 10;
}

void
randsig_source_ff::set_offset (float offset)
{
  d_offset = offset;
}
```

Filters:

filter_impulse_test.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import blks
from gnuradio.eng_option import eng_option
#import sys
#import math

class filter_impulse_test (gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)


        # build graph
```

```
        zeroes = [0 for i in range(999)]
        impulse = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
        impulse.extend(zeroes)

        src = gr.vector_source_f(impulse, 0)

        filter_coeffs = optfir.low_pass (1,           # gain
                                         32000,   # sampling rate
                                         340,        # passband cutoff
                                         1000,       # stopband cutoff
                                         0.1,         # passband ripple
                                         60)          # stopband
attenuation

        filter = gr.fir_filter_fff (1, filter_coeffs)

        # file as final sink
        file_sink = gr.file_sink(gr.sizeof_float, "./impulse_test.dat")

        # now wire it all together
        self.connect (src, filter, file_sink)



if __name__ == '__main__':
    fg = filter_impulse_test()
    try:
        fg.run()
    except KeyboardInterrupt:
        pass
```

filter_response_test.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import blks
from gnuradio.eng_option import eng_option
#import sys
#import math

class filter_response_test (gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)


        # build graph

        src = gr.noise_source_f(gr.GR_GAUSSIAN, 10, 10)

        filter_coeffs = optfir.low_pass (1,           # gain
                                         32000,   # sampling rate
                                         340,        # passband cutoff
```

```
                                            1000,       # stopband cutoff
                                            0.1,         # passband ripple
                                            60)          # stopband
attenuation

        filter = gr.fir_filter_fff (1, filter_coeffs)

        # file as final sink
        file_sink = gr.file_sink(gr.sizeof_float, "./response_test.dat")

        # now wire it all together
        self.connect (src, filter, file_sink)



if __name__ == '__main__':
    fg = filter_response_test()
    try:
        fg.run()
    except KeyboardInterrupt:
        pass
```

fpgatest_failed.py:

```
#!/usr/bin/env python
#
# Copyright 2004,2005 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_option import eng_option
from optparse import OptionParser

class my_graph(gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)
```

```
        parser = OptionParser(option_class=eng_option)
        parser.add_option("-O", "--audio-output", type="string", default="",
                          help="pcm output device name.  E.g., hw:0,0 or
/dev/dsp")
        parser.add_option("-r", "--sample-rate", type="eng_float",
default=32000,
                          help="set sample rate to RATE (32000)")
        (options, args) = parser.parse_args ()
        if len(args) != 0:
            parser.print_help()
            raise SystemExit, 1

        sample_rate = int(options.sample_rate)
        ampl = 1

        src1 = gr.sig_source_f (sample_rate,gr.GR_SIN_WAVE,440,ampl,0)
        src2 = gr.sig_source_f (sample_rate,gr.GR_SIN_WAVE,320,ampl,0)
        src3 = gr.sig_source_f (sample_rate,gr.GR_SIN_WAVE,650,ampl,0)
        sum = gr.add_ff()
        src = gr.add_const_ff(1)
        self.connect (src1, (sum, 0))
        self.connect (src2, (sum, 1))
        self.connect (src3, (sum, 2))
        self.connect (sum, src)

        expanding = gr.multiply_const_ff(65535)

        FPGA_IN = gr.file_sink(gr.sizeof_float, "/dev/MQP_Pipe")

        FPGA_OUT = gr.file_source(gr.sizeof_float, "/dev/MQP_Pipe")

        unexpanding = gr.multiply_const_ff(1/65535)

        file_sink = gr.file_sink(gr.sizeof_float, "./filter_results.dat")

        self.connect (src, expanding, FPGA_IN)
        self.connect (FPGA_OUT, unexpanding, file_sink)



if __name__ == '__main__':
    try:
        my_graph().run()
    except KeyboardInterrupt:
        pass
```

FPGA_frequency_response.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import blks
from gnuradio.eng_option import eng_option
#import sys
#import math
```

```
class filter_response_test (gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)


        # build graph

        src = gr.noise_source_f(gr.GR_GAUSSIAN, 10, 10)

        FPGA_OUT = gr.file_source(gr.sizeof_float, "/dev/MQP_Pipe")

        FPGA_IN = gr.file_sink (gr.sizeof_float, "/dev/MQP_Pipe")

        scale = gr.multiply_const_ff(1.0/65535.0)

        # file as final sink
        file_sink = gr.file_sink(gr.sizeof_float, "./FPGA_response_test.dat")

        # now wire it all together
        self.connect (src, FPGA_IN)
        self.connect (FPGA_OUT, scale, file_sink)

if __name__ == '__main__':
    fg = filter_response_test()
    try:
        fg.run()
    except KeyboardInterrupt:
        pass
```

FPGA_impulse_test.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import blks
from gnuradio.eng_option import eng_option
#import sys
#import math

class filter_response_test (gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)


        # build graph

                zeroes = [0 for i in range(500)]
        impulse = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0]
        impulse.extend(zeroes)

        src = gr.vector_source_f(impulse, 0)
```

```
        FPGA_OUT = gr.file_source(gr.sizeof_float, "/dev/MQP_Pipe")

        FPGA_IN = gr.file_sink (gr.sizeof_float, "/dev/MQP_Pipe")

        scale = gr.multiply_const_ff(1.0/65535.0)

        # file as final sink
        file_sink = gr.file_sink(gr.sizeof_float, "./FPGA_response_test.dat")

        # now wire it all together
        self.connect (src, FPGA_IN)
        self.connect (FPGA_OUT, scale, file_sink)

if __name__ == '__main__':
    fg = filter_response_test()
    try:
        fg.run()
    except KeyboardInterrupt:
        pass
```

usrp_wfm_rcv_filt_nogui.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
#import usrp_dbid disabled because YunLing complains
import sys
import math

#def pick_subdevice(u):
    """
    The user didn't specify a subdevice on the command line.
    Try for one of these, in order: TV_RX, BASIC_RX, whatever is on side A.

    @return a subdev_spec
    """
#    return usrp.pick_subdev(u, (usrp_dbid.TV_RX,
#                                usrp_dbid.TV_RX_REV_2,
#                                usrp_dbid.BASIC_RX))


class wfm_rx_graph (gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)

        parser=OptionParser(option_class=eng_option)
        parser.add_option("-R", "--rx-subdev-spec", type="subdev",
default=None,
                          help="select USRP Rx side A or B (default=A)")
        parser.add_option("-f", "--freq", type="eng_float", default=100.1e6,
```

```
                          help="set frequency to FREQ", metavar="FREQ")
        parser.add_option("-g", "--gain", type="eng_float", default=None,
                          help="set gain in dB (default is midpoint)")
        parser.add_option("-O", "--audio-output", type="string", default="",
                          help="pcm device name.  E.g., hw:0,0 or surround51
or /dev/dsp")

        (options, args) = parser.parse_args()
        if len(args) != 0:
            parser.print_help()
            sys.exit(1)

        self.vol = 10
        self.state = "FREQ"
        self.freq = 0

        # build graph

        self.u = usrp.source_c()                    # usrp is data source

        adc_rate = self.u.adc_rate()                # 64 MS/s
        usrp_decim = 200
        self.u.set_decim_rate(usrp_decim)
        usrp_rate = adc_rate / usrp_decim           # 320 kS/s
        chanfilt_decim = 1
        cleanup_decim = 1
        demod_rate = usrp_rate / chanfilt_decim
        audio_decimation = 10
        audio_rate = demod_rate / audio_decimation  # 32 kHz


        #if options.rx_subdev_spec is None:
        #    options.rx_subdev_spec = pick_subdevice(self.u)

        self.u.set_mux(usrp.determine_rx_mux_value(self.u,
options.rx_subdev_spec))
        self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
        print "Using RX d'board %s" % (self.subdev.side_and_name(),)


        chan_filt_coeffs = optfir.low_pass (1,           # gain
                                            usrp_rate,   # sampling rate
                                            80e3,        # passband cutoff
                                            115e3,       # stopband cutoff
                                            0.1,         # passband ripple
                                            60)          # stopband
attenuation
        #print len(chan_filt_coeffs)
        chan_filt = gr.fir_filter_ccf (chanfilt_decim, chan_filt_coeffs)

        self.guts = blks.wfm_rcv (self, demod_rate, audio_decimation)

        cleanup_coeffs = optfir.low_pass (1,           # gain
                                            usrp_rate,   # sampling rate
                                            340,         # passband cutoff
                                            800,       # stopband cutoff
                                            0.1,         # passband ripple
```

```
                                          60)          # stopband
attenuation

        cleanup = gr.fir_filter_fff (cleanup_decim, cleanup_coeffs)

        self.volume_control = gr.multiply_const_ff(self.vol)

        # sound card as final sink
        #audio_sink = audio.sink(int(audio_rate),
        #                        options.audio_output,
        #                        False)  # ok_to_block

        # file as final sink
        file_sink = gr.file_sink(gr.sizeof_float, "./filtered_reception.dat")

        # now wire it all together
        self.connect (self.u, chan_filt, self.guts, cleanup,
self.volume_control, file_sink)


        if options.gain is None:
            # if no gain was specified, use the mid-point in dB
            g = self.subdev.gain_range()
            options.gain = float(g[0]+g[1])/2

        if abs(options.freq) < 1e6:
            options.freq *= 1e6

        # set initial values

        self.set_gain(options.gain)

        if not(self.set_freq(options.freq)):
            self._set_status_msg("Failed to set initial frequency")

    def set_vol (self, vol):
        self.vol = vol
        self.volume_control.set_k(self.vol)
        self.update_status_bar ()

    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital down converter.
        """
        r = self.u.tune(0, self.subdev, target_freq)

        if r:
            self.freq = target_freq
            self.update_status_bar()
```

```
            self._set_status_msg("OK", 0)
            return True

        self._set_status_msg("Failed", 0)
        return False

    def set_gain(self, gain):
        self.subdev.set_gain(gain)

    def update_status_bar (self):
        msg = "Freq: %s  Volume:%f  Setting:%s" % (
            eng_notation.num_to_str(self.freq), self.vol, self.state)
        self._set_status_msg(msg, 1)

    def _set_status_msg(self, msg, which=0):
        print msg


if __name__ == '__main__':
    fg = wfm_rx_graph()
    try:
        fg.run()
    except KeyboardInterrupt:
        pass
```

wfm_tx_multisignal.py:

```
#!/usr/bin/env python

"""
Transmit N simultaneous narrow band FM signals.

They will be centered at the frequency specified on the command line,
and will spaced at 25kHz steps from there.

The program opens N files with names audio-N.dat where N is in [0,7].
These files should contain floating point audio samples in the range [-1,1]
sampled at 32kS/sec.  You can create files like this using
audio_to_file.py
"""

from gnuradio import gr, eng_notation
from gnuradio import usrp
from gnuradio import audio
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import usrp_dbid
import math
import sys

from gnuradio.wxgui import stdgui, fftsink
from gnuradio import tx_debug_gui
import wx
```

```
######################################################
# instantiate one transmit chain for each call

class pipeline(gr.hier_block):
    def __init__(self, fg, lo_freq, audio_rate, if_rate):

        src1 = gr.sig_source_f (if_rate,gr.GR_SIN_WAVE,1440,100,0)
        src2 = gr.sig_source_f (if_rate,gr.GR_SIN_WAVE,320,100,0)
        src3 = gr.sig_source_f (if_rate,gr.GR_SIN_WAVE,2650,100,0)
        sum = gr.add_ff()
        src = gr.add_const_ff(1)
        fg.connect (src1, (sum, 0))
        fg.connect (src2, (sum, 1))
        fg.connect (src3, (sum, 2))
        fg.connect (sum, src)

        fmtx = blks.nbfm_tx (fg, audio_rate, if_rate,
                             max_dev=5e3, tau=75e-6)

        # Local oscillator
        lo = gr.sig_source_c (if_rate,        # sample rate
                              gr.GR_SIN_WAVE, # waveform type
                              lo_freq,        #frequency
                              1.0,            # amplitude
                              0)              # DC Offset
        mixer = gr.multiply_cc ()

        fg.connect (src, fmtx, (mixer, 0))
        fg.connect (lo, (mixer, 1))

        gr.hier_block.__init__(self, fg, src, mixer)



class fm_tx_graph (stdgui.gui_flow_graph):
    def __init__(self, frame, panel, vbox, argv):
        stdgui.gui_flow_graph.__init__ (self, frame, panel, vbox, argv)

        parser = OptionParser (option_class=eng_option)
        parser.add_option("-T", "--tx-subdev-spec", type="subdev",
default=None,
                          help="select USRP Tx side A or B")
        parser.add_option("-f", "--freq", type="eng_float", default=None,
                           help="set Tx frequency to FREQ [required]",
metavar="FREQ")
        parser.add_option("","--debug", action="store_true", default=False,
                          help="Launch Tx debugger")
        (options, args) = parser.parse_args ()

        if len(args) != 0:
            parser.print_help()
            sys.exit(1)

        if options.freq is None:
            sys.stderr.write("fm_tx4: must specify frequency with -f FREQ\n")
            parser.print_help()
            sys.exit(1)
```

```
        # ---------------------------------------------------------------
        # Set up constants and parameters

        self.u = usrp.sink_c ()       # the USRP sink (consumes samples)

        self.dac_rate = self.u.dac_rate()                    # 128 MS/s
        self.usrp_interp = 400
        self.u.set_interp_rate(self.usrp_interp)
        self.usrp_rate = self.dac_rate / self.usrp_interp    # 320 kS/s
        self.sw_interp = 10
        self.audio_rate = self.usrp_rate / self.sw_interp    # 32 kS/s

        # determine the daughterboard subdevice we're using
        if options.tx_subdev_spec is None:
            options.tx_subdev_spec = usrp.pick_tx_subdevice(self.u)

        m = usrp.determine_tx_mux_value(self.u, options.tx_subdev_spec)
        #print "mux = %#04x" % (m,)
        self.u.set_mux(m)
        self.subdev = usrp.selected_subdev(self.u, options.tx_subdev_spec)
        print "Using TX d'board %s" % (self.subdev.side_and_name(),)

        self.subdev.set_gain(self.subdev.gain_range()[1])    # set max Tx
gain
        self.set_freq(options.freq)
        self.subdev.set_enable(True)                         # enable
transmitter

        sum = gr.add_cc ()

        # Instantiate channel
        t = pipeline (self, 0, self.audio_rate, self.usrp_rate)
        self.connect (t, (sum, 0))

        gain = gr.multiply_const_cc (4000.0)

        # connect it all
        self.connect (sum, gain)
        self.connect (gain, self.u)

        # plot an FFT to verify we are sending what we want
        if 1:
            post_mod = fftsink.fft_sink_c(self, panel, title="Post
Modulation",
                                          fft_size=512,
sample_rate=self.usrp_rate,
                                          y_per_div=20, ref_level=40)
            self.connect (sum, post_mod)
            vbox.Add (post_mod.win, 1, wx.EXPAND)


        if options.debug:
            self.debugger = tx_debug_gui.tx_debug_gui(self.subdev)
            self.debugger.Show(True)
```

```
    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.  Finally, we feed
        any residual_freq to the s/w freq translater.
        """

        r = self.u.tune(self.subdev._which, self.subdev, target_freq)
        if r:
            print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
            print "r.dxc_freq      =", eng_notation.num_to_str(r.dxc_freq)
            print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
            print "r.inverted      =", r.inverted

            # Could use residual_freq in s/w freq translator
            return True

        return False

def main ():
    app = stdgui.stdapp (fm_tx_graph, "Single-channel FM Tx")
    app.MainLoop ()

if __name__ == '__main__':
    main ()
```

Other useful programs:

float_to_ascii.py:

```
import sys, struct, os

if len(sys.argv) < 2 and len(sys.argv) > 3:
    print "Usage: %s <input file> <optional output file>" % sys.argv[0]

data_file = open(sys.argv[1], #sys.argv[1] = first thing on the command line
                 mode='rb') #rb = Read Binary file
out_file = None

try:
    if len(sys.argv) == 3:
        out_file = open(sys.argv[2], 'w') #Open a file for writing
```

```
    while True:
        raw = data_file.read(4) #Read 4 bytes
        if len(raw) < 4:
            break #We've reached the end of the file
        f = struct.unpack('f', raw)[0] #Convert bytes to a float
        if out_file is None:
            print f #Print the new value
        else:
            out_file.write("%s\n" % (f)) #Print f as a string to the file
finally:
    if data_file is not None:
        data_file.close()
    if out_file is not None:
        out_file.close()
```

whitenoise_gen.py:

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import blks
from gnuradio.eng_option import eng_option
#import sys
#import math

class whitenoise_gen (gr.flow_graph):

    def __init__(self):
        gr.flow_graph.__init__(self)


        # build graph

        src = gr.noise_source_f(gr.GR_GAUSSIAN, 10, 10)

        # file as final sink
        file_sink = gr.file_sink(gr.sizeof_float, "./white_noise.dat")

        # now wire it all together
        self.connect (src, file_sink)


if __name__ == '__main__':
    fg = whitenoise_gen()
    try:
        fg.run()
    except KeyboardInterrupt:
        pass
```

HDL and project files will be included as an e-appendicie via the E-submission system.